EXPRESS MAIL NO. EL029404528US

PATENT
Attorney Docket No. 00-8013

SYSTEM AND METHOD FOR AUTOMATIC LOADING OF AN XML DOCUMENT DEFINED BY A DOCUMENT-TYPE DEFINITION INTO A RELATIONAL DATABASE INCLUDING THE GENERATION OF A RELATIONAL SCHEMA THEREFOR

BACKGROUND OF THE INVENTION

Related Applications

[0001] This application claims priority of U.S. Provisional Application No. 60/182,939 entitled "METHOD AND APPARATUS FOR AUTOMATIC LOADING OF XML DOCUMENTS INTO RELATIONAL DATABASES," filed February 16, 2000.

Field of the Invention

[0002] The invention relates to a method and system for automatically loading an extensible markup language (XML) document, as validated by a document-type definition (DTD), into a relational database.

Description of the Related Art

[0003] Touted as the ASCII of the future, eXtensible Markup Language (XML) is used to define markups for information modeling and exchange in many industries. By enabling automatic data flow between businesses, XML is contributing to efforts that are pushing the world into the electronic commerce (e-commerce) era. It is envisioned that collection, analysis, and management of XML data will be tremendously important tasks for the era of e-commerce. XML data, i.e., data surrounded by an initiating tag (e.g., `<tag>`) and a terminating tag (e.g., `</tag>`) can be validated by a document-type definition (DTD) as will be hereinafter described. As can be seen, boldface text is used to describe XML and DTD contents as well as names for table and document tags and fields.

[0004] Some background on XML and DTDs may be helpful in understanding the difficulties present in importing XML data into a relational database. XML is currently used both for defining document markups (and, thus, information modeling) and for data exchange. XML documents are composed of character data and nested tags used to document semantics of the embedded text. Tags can be used freely in an XML document (as long as their use conforms to the XML specification) or can be used in accordance

with document-type definitions (DTDs) for which an XML document declares itself in conformance. An XML document that conforms to a DTD is referred to as a valid XML document.

[0005] A DTD is used to define allowable structures of elements (i.e., it define allowable tags, tag structure) in a valid XML document. A DTD can basically include four kinds of declarations: element types, attribute lists, notations, and entity declarations.

[0006] An element type declaration is analogous to a data type definition; it names an element and defines the allowable content and structure. An element may contain only other elements (referred to as element content) or may contain any mix of other elements and text, one such mixed content is represented as **PCDATA**. An **EMPTY** element type declaration is used to name an element type without content (it can be used, for example, to define a placeholder for attributes). Finally, an element type can be declared with content ANY meaning the type (content and structure) of the element is arbitrary.

[0007] Attribute-list declarations define attributes of an element type. The declaration includes attribute names, default values and types, such as **CDATA**, **NOTATION**, and **ENUMERATION**. Two special types of attributes, **ID** and **IDREF**, are used to define references between elements. An **ID** attribute is used to uniquely identify the element; an **IDREF** attribute can be used to reference that element (it should be noted that an **IDREFS** attribute can reference multiple elements). **ENTITY** declarations facilitate flexible organization of XML documents by breaking the documents into multiple storage units. A **NOTATION** declaration identifies non-XML content in XML documents. It is assumed herein that one skilled in the art of XML documents that include a DTD is familiar with the above terminology.

[0008] Element and attribute declarations define the structure of compliant XML documents and the relationships among the embedded XML data items. **ENTITY** declarations, on the other hand, are used for physical organization of a DTD or XML document (similar to macros and inclusions in many programming languages and word processing documents). For purposes of the present invention, it has been assumed that entity declarations can be substituted or expanded to give an equivalent DTD with only

element type and attribute-list declarations, since they do not provide information pertinent to modeling of the data (this can be referred to as a logical DTD). In the discussion that follows, DTD is used to refer to a logical DTD. The logical DTD in Example 1 below (for books, articles and authors) is used throughout for illustration.

Example 1 DTD for Books , Articles, and Authors.

```
<!ELEMENT book (booktitle, (author*|editor))>
<!ELEMENT booktitle (#PCDATA)>
<!ELEMENT article (title, (author, affiliation?)+, contactauthors?)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT contactauthors EMPTY>
<!ATTLIST contactauthors authorIDs IDREFS #REQUIRED>
<!ELEMENT monograph (title, author, editor)>
<!ELEMENT editor (book|monograph)*>
<!ATTLIST editor name CDATA #IMPLIED>
<!ELEMENT author (name)>
<!ATTLIST author id ID #REQUIRED>
<!ELEMENT name (firstname?, lastname)>
<!ELEMENT firstname (#PCDATA)>
<!ELEMENT lastname (#PCDATA)>
<!ELEMENT affiliation ANY>
```

[0009] The task of developing a relational schema for XML documents requires understanding the components of, and relationships within, such documents. A DTD defines a structure for XML documents that can be seen as an ordered graph composed of element type declarations. A DTD has the following properties (also referred to herein as data and/or content particles):

[0010] *Grouping*: Within an element type definition, elements that are associated within parentheses participate in a grouping relationship, and are defined as a group. This relationship can be further classified as sequence grouping (wherein the elements are separated by a comma ',') or choice grouping (wherein the elements are separated by a vertical bar '|') according to the operator used as a delimiter in the grouping.

[0011] *Nesting*: An element type definition provides the mechanism for modeling relationships that can be represented structurally as a hierarchy of elements. These relationships are referred to as a nesting relationship (a structural view can be chosen to avoid having to choose particular semantics for all such relationships).

[0012] *Schema Ordering*: The logical ordering among element types specified in a DTD. For example, line one of the DTD in Example 1 specifies that in a `book`, a `booktitle` precedes a list of `authors` (or an `editor`).

[0013] *Existence*: In a DTD, an element type with no content declares the existence of an element with no structure or value for that element type. This kind of virtual element is declared so that attributes can be defined for unique properties of the element or for element referencing relationships.

[0014] *Occurrence*: Occurrence indicators (i.e., "`?`", "`*`", and "`+`") indicate optional occurrence or repetition of a content particle in an element definition. For example, in Example 1, the grouping (`book | monograph`) can appear zero or more times in an editor element which is indicated by the "`*`" following this grouping in line eight of the DTD in Example 1.

[0015] *Element Referencing*: Elements reference one another using attributes of type `ID` and `IDREF`(s). For example, in Example 1, `contactauthors` has an element reference relationship with `author`.

[0016] These properties and relationships illustrate that a DTD not only defines element types for conforming XML documents, but also provides valuable information about the structure of an XML document and relationships between its elements. Besides that, while the DTD specifies a schema ordering between element types thereof, XML documents also have a physical ordering, or data ordering, of data elements. The challenges of mapping a DTD to a relational database model arise from a mismatch between (1) types of properties and relationships embedded in DTDs and (2) those modeled by relational database models.

[0017] Turning to the task of loading XML data (as validated by a DTD) into a relational database, the prior art in database systems must be considered. Database systems are traditional, well-known tools for managing data. After many years of development, database technology has matured and contributed significantly to the rapid growth of business and industry. Relational database systems are a proven technology for managing business data and are used everywhere by various sizes of companies for their critical

business tasks. Commercial relational database products embody years of research and development in modeling, storage, retrieval, update, indexing, transaction processing, and concurrency control, and continue to add capabilities to address new kinds of data, such as multimedia.

[0018] With more and more data flowing in XML formats, there have been attempts to extend prior art relational database systems to accommodate XML data. Such an approach has avoided re-inventing database technology to suit XML data but, more importantly, takes best advantage of the power of relational database technology and the wealth of experience in optimizing and using the technology.

[0019] There have been several problems to overcome in bringing XML data into a relational database for management, including defining a relational schema for the XML data, loading the XML data into a relational database, and transforming XML queries (whether formulated in extensible stylesheet language [XSL], XML Query Language [XML-QL] or other XML query standards) into meaningful structured query language [SQL] queries.

[0020] Prior attempts to solve these problems have fallen short of an efficient and, preferably automatic, way to import XML data into a relational database schema. Current industry enterprise database management system (DBMS) vendors, such as DB2 and Oracle 8i, provide XML extensions for bringing XML data into a relational database. However, these methods are far from automatic. These vendors require users to manually design the relational schema for a given DTD and to define the mapping between the DTD and the user-designed schema for the loading of XML documents. While this manual approach can be straightforward, and these vendors provide tools to assist the user, users must be very familiar with the XML data, the DTD therefor and the particular database system used.

[0021] In addition, the prior art approach only works well for generating a relatively simple relational schema, and is not effective or efficient when the data contains complex relationships between tables. It is most appropriate for a small number of short or familiar DTDs. The known approach also requires experts on both XML and relational

techniques. For more complex DTDs and a more robust relational schema, the manual approach becomes more difficult and requires specialized expertise. The common existence of non-straightforward relational database schema cases requires a more advanced approach to generating the relational schema and defining the load mapping definition.

[0022] Other attempts to conceive a method that automatically loads XML data into a relational database have also proven to be of limited success. In accordance with these failed attempts, the user is required to either mine XML documents or to simplify the DTD. In either case, semantics captured by the DTD or XML documents are lost, e.g., how elements may be grouped within the data or as defined by the DTD, and how to distinguish between DTD specific symbols, such as * and +, etc.

[0023] Other prior art attempts to load XML data into a relational database schema include one method by which a large amount of data is placed into relational databases by creating a relational data schema and applying data mining over a large number of the same type of XML documents, and then abstracting a relational data schema out of them. Then, the data is loaded into the relational tables. For parts that cannot fit into the relational table schema, an overflow graph is generated.

[0024] Others have done benchmark testing on a relational schema generated out of XML data by four variations of a basic mapping approach -- but this work does not consider or does not require a DTD.

[0025] However, one known form of benchmark testing that does consider a DTD was performed by Shanmugasundaram et. al. who investigated schema conversion techniques for mapping a DTD to a relational schema by giving (and comparing) three alternative methods based on traversal of a DTD tree and creation of element trees. This process simplified the DTDs and then mapped those into a relational schema. While this approach also works with simple XML data structures, portions of the structure, properties and embedded relationships among elements in XML documents are often lost.

[0026] Instead of bringing the semi-structured data into the relational model, there are other approaches to bring the XML data into semi-structured, object-oriented, or object-

relational database management systems (DBMS). Some commercial rational DBMSs, e.g., IBM's DB2 and Oracle, have begun to incorporate XML techniques into their databases, e.g., IBM Alphaworks visual XML tools, IBM DB2 XML Extender, and Oracle 8i.

[0027] Recently IBM's Alphaworks proposed a new set of visual XML tools that can visually create and view DTDs and XML documents. In its tools, IBM has proposed the idea of breaking DTDs into elements, notations, and entities that use components grouped with properties sequential, choice, attribute, and relationship and with repetition properties to construct DTDs. Tools to do XML translation and XML generation from SQL queries are provided. However, a method by which to load the XML data into relational tables was not addressed in this prior art.

[0028] The DB2 XML Extender can store, compose and search XML documents. Document storing is accomplished in two ways. First, the XML document is stored as a whole for indexing, referred to as way of storing the XML document as an XML column. Second, pieces of XML data are stored into table(s), referred to as XML collections.

[0029] Oracle 8i is an XML-enabled database server that can do XML document reading and writing in the document's native format. An XML document is stored as data and distributed across nested-relational tables. This XML SQL utility provides a method to load the XML documents in a canonical form into a preexisting schema that users manually (and previously) designed.

SUMMARY OF THE INVENTION

[0030] According to the invention, a relational schema definition is examined for XML data, a relational schema is created out of a DTD, and XML data is loaded into the generated relational schema that adheres to the DTD. In this manner, the data semantics implied by the XML are maintained so that more accurate and efficient management of the data can be performed.

[0031] Starting with a DTD for an XML document containing data (rather than analyzing the relationships between the actual elements of the XML data), all of the information in the DTD is captured into metadata tables, and then the metadata tables are queried to

generate the relational schema. Then, the data contained in the XML document can be loaded into the generated relational schema. This method can be described by three broad steps:

[0032] First, the DTD is stored as metadata (i.e., a transformation and/or recasting of the data contained in the DTD) in tables -- i.e., the metadata is used to describe the information of the DTD associated with the XML documents. This approach provides flexibility to manipulate the DTD by standard SQL queries.

[0033] Second, a relational schema is generated from the metadata stored in the metadata tables. This step provides additional flexibility in that, although the relational schema can be directly mapped from the metadata tables according to the invention, the metadata tables can also be queried to do optimizing or restructuring on the metadata tables representative of the XML data structure stored in the DTD.

[0034] Third, data contained in the XML document is loaded into the tables as defined by the relational schema (which is generated in the previous step), by using the associated metadata tables.

[0035] According to the invention, the inventive metadata-driven approach includes the following beneficial characteristics:

[0036] *For Storing*: All of the information contained in the DTD is captured in the metadata tables. It is anticipated that the XML document and the DTD can be reconstructed from the relational data and metadata as needed.

[0037] *For Mapping*: The generated relational schema is rich in relationships that are useful in processing queries.

[0038] *For Loading*: Mappings between the XML document and the final relational schema are captured in the metadata tables. It is contemplated that the relational data can be synchronized with XML data, which means whenever there is a data update in the relational data, the effect is also reflected in the XML data, and vice versa.

[0039] In one aspect, the invention relates to a method for generating a schema for a relational database corresponding to a document having a document-type definition and data complying with the document-type definition. The document-type definition has

content particles representative of the structure of the document data including one or more of the following content particles: elements, attributes of elements, nesting relationships between elements, grouping relationships between elements, schema ordering indicators, existence indicators, occurrence indicators and element ID referencing indicators. The method also contemplates loading the data into the relational database in a manner consistent with the relational schema.

[0040] The method comprises the steps of: extracting metadata from the document-type definition representative of the document-type definition; generating the schema for the relational database from the metadata, wherein at least one table is thereby defined in the relational database corresponding to at least one content particle of the document-type definition via the metadata, and at least one column is defined in each of the at least one table corresponding to another of at least one content particle of the document-type definition; and loading the document data into the at least one table of the relational database according to the relational schema.

[0041] The extracting step of the inventive method can further comprise the steps of: generating an item metadata table corresponding to element type content particles in the document-type definition; creating at least one default item in the item metadata table; generating a row in the item metadata table corresponding to each of the element type content particles of the document-type definition; generating an attribute metadata table corresponding to attribute type content particles in the document-type definition; creating a default attribute value in the attribute metadata table corresponding to any default items in the item metadata table; generating a row in the attribute metadata table corresponding to each of the attribute type content particles of each element type stored in the item metadata table; generating a nesting metadata table corresponding to nesting relationship content particles in the document-type definition; and generating a row in the nesting metadata table corresponding to each relationship between items identified in the item metadata table.

[0042] In some embodiments of the invention, the generated nesting table row can indicate the cardinality between a pair of items. The cardinality can be one of one-to-one

and one-to-many. The generated nesting table row can indicate a relationship between a parent item and a child item. The generated nesting table row can indicate the position of the child item in a definition of the parent item.

[0043] In other embodiments of the invention, the generating step can further comprise the steps of: creating a table in the schema of the relational database corresponding to each row of the metadata item table; generating at least one default field in the table of the schema; altering the schema of the relational database to add a column to each table in the schema corresponding to each row of the metadata attribute table related to the particular metadata item table row; altering the tables in the schema of the relational database to add links between tables in the schema corresponding to a relationship identified in each row of the metadata nesting table; altering the tables in the schema of the relational database by adding a foreign key to a parent table if the identified relationship is a one-to-one relationship; and altering the tables in the schema of the relational database by adding a foreign key to a child table if the identified relationship is a one-to-many relationship.

[0044] In additional embodiments of this invention, the loading step can further comprise the steps of: initializing a link table; determining whether each item in the metadata nesting table contains a group type; initializing a pattern-mapping table; directly mapping a link into the link table for each item in the metadata nesting table that does not contain a group type; creating an additional link table containing a mapping of a link pattern for each group type identified in the metadata item table; retrieving a preselected set of rows corresponding to each item in the metadata item table; mapping a create tuple loading action in the pattern mapping table corresponding to each item in the item metadata table; mapping an update tuple loading action in the pattern mapping table corresponding to each attribute in the attribute metadata table; mapping a create tuple loading action in the pattern mapping table corresponding to each group in a link; mapping an assign action tuple loading action in the pattern mapping table corresponding to each pair in the same link corresponding to each link in the link pattern table; and forming a tree structure with

the document data; and traversing the formed tree and updating the at least one relational database table according to the rows of the pattern mapping table.

[0045] In yet further embodiments of this invention, the method can also comprise the step of optimizing the metadata. This optimizing step can further comprise the steps of: eliminating duplicate particle references in the metadata; and simplifying references to corresponding elements, links and attributes in the metadata.

[0046] In other aspects of the invention, a system is provided for generating a schema for a relational database corresponding to a document having a document-type definition and data complying with the document-type definition and loading the data into the relational database in a manner consistent with the relational schema.

[0047] The key inventive features of the system include an extractor adapted to read a document-type definition that extracts metadata from the document-type definition representative of the document-type definition; a generator operably interconnected to the extractor for generating the schema for the relational database from the metadata, wherein at least one table is thereby defined in the relational database corresponding to at least one content particle of the document-type definition via the metadata, and at least one column is defined in each of the tables corresponding to another content particle of the document-type definition; and a loader operably interconnected to the generator for loading the document data into the table(s) of the relational database according to the relational schema.

[0048] In various embodiments of the extractor for the system, the extractor can generate an item metadata table corresponding to element type content particles in the document-type definition. The extractor can create at least one default item in the item metadata table. The extractor can generate a row in the item metadata table corresponding to each of the element type content particles of the document-type definition. The extractor can generate an attribute metadata table corresponding to attribute type content particles in the document-type definition. The extractor can generate a row in the attribute metadata table corresponding to each of the attribute type content particles of each element type stored in the item metadata table. The extractor can generate a nesting metadata table

corresponding to nesting relationship content particles in the document-type definition. The extractor can generate a row in the nesting metadata table corresponding to each relationship between items identified in the item metadata table.

[0049] In various embodiments of the generator for the system described herein, the generator can create a table in the schema of the relational database corresponding to each row of the metadata item table. The generator can alter the schema of the relational database to add a column to each table in the schema corresponding to each row of the metadata attribute table related to the particular metadata item table row. The generator can alter the tables in the schema of the relational database to add links between tables in the schema corresponding to a relationship identified in each row of the metadata nesting table. The generator can alter the tables in the schema of the relational database by adding a foreign key to a parent table if a relationship identified between a pair of tables is a one-to-one relationship. The generator can alter the tables in the schema of the relational database by adding a foreign key to a child table if a relationship identified between a pair of tables is a one-to-many relationship.

[0050] In various aspects of the loader of this system, the loader can initialize a link table and/or a pattern-mapping table. The loader can determine whether each item in the metadata nesting table contains a group type content particle. The loader can directly map a link into the link table for each item in the metadata nesting table that does not contain a group type. The loader can create an additional link table containing a mapping of a link pattern for each group type identified in the metadata item table. The loader can retrieve a preselected set of rows corresponding to each item in the metadata item table. The loader can map a create tuple loading action in the pattern mapping table corresponding to each item in the item metadata table. The loader can map an update tuple loading action in the pattern mapping table corresponding to each attribute in the attribute metadata table. The loader can map a create tuple loading action in the pattern mapping table corresponding to each group in a link; and map an assign action tuple loading action in the pattern mapping table corresponding to each pair in the same link corresponding to each link in the link pattern table. The loader can form a tree structure

with the document data. The loader can traverse the formed tree and update the relational database table(s) according to the rows of the pattern mapping table.

[0051] In other embodiments of the system, the system can also be provided with an optimizer for refining the metadata. The optimizer can eliminate duplicate particle references in the metadata. The optimizer can simplify references to corresponding elements, links and attributes in the metadata.

[0052] In other embodiments of the invention, the document can be an XML document. The document-type definition can be a DTD. The data can be tagged data.

BRIEF DESCRIPTION OF THE DRAWINGS

In the drawings:

[0053] Fig. 1 is a schematic view detailing a system for generating a relational schema from a document type definition, forming a relational database from the relational schema and loading the contents of an extensible document into the relational database according to the relational schema.

[0054] Fig. 1A is a diagrammatic representation of the system of Fig. 1 showing the extraction of a document-type definition from an extensible document, the generation of a relational schema therefrom and the loading of data contained in the extensible document into the relational database.

[0055] Fig. 1B is a schematic representation of interaction between tables created in the system and method shown in Figs. 1 and 1A.

[0056] Fig. 2 is a flowchart detailing three broad method steps according to the invention schematically shown in Figs. 1, 1A and 1B, namely, storing the document-type definition into metadata tables, creating a relational database table schema from the metadata of the metadata tables, and loading data contained in an extensible document into tables contained in the formed relational schema.

[0057] Fig. 3 is a flowchart detailing the step for storing the document-type definition into metadata tables shown in Fig. 2 in which a metadata item table is created and filled with element-types declared in the document-type definition.

[0058] Fig. 4 is a flowchart detailing another step of the method shown in Fig. 2 of storing the document-type definition information into metadata tables in which a metadata attribute table is created and filled with attributes defined in the document-type definition.

[0059] Fig. 5 is a flowchart showing another step of the method shown in Fig. 2 of storing the document-type definition table into metadata tables including the step of building groups and forming a metadata nesting table from the metadata item table formed in Fig. 3.

[0060] Fig. 5A is a flow chart detailing a portion of the flow chart of Fig. 5 corresponding to the steps to be performed when an element type is encountered during generation of the metadata nesting table.

[0061] Fig. 6 is a flowchart detailing the step of the method of Fig. 2 in which a relational database schema is generated from the metadata tables including the step of forming tables for each element type in the metadata item table formed in Fig. 3 with default fields provided therein.

[0062] Fig. 7 is a flowchart detailing another method step from the method shown in Fig. 2 in which tables are created to form a relational table schema from the metadata tables in which columns are added to the tables formed in the method shown in Fig. 6 for each of the attributes defined in the metadata attribute tables formed in Fig. 4.

[0063] Fig. 8 is a flowchart detailing a method step corresponding additionally to the method step in Fig. 2 in which a relational schema is created from the metadata tables in which nesting relationships are determined between the attributes of the various tables and index columns are added to the various tables in the schema corresponding to those nesting relationships identified in the method step in Fig. 2 in which the metadata nesting table is constructed.

[0064] Fig. 9 is a flowchart corresponding to the initialization of the pattern mapping table as described in Figs. 1, 1A, 1B and 2.

[0065] Fig. 10 is a flowchart corresponding to the method step of Fig. 9 in which at least one link table is initialized.

[0066] Fig. 11 is a flowchart corresponding to a method step in Fig. 9 in which the pattern mapping table is initialized from the data contained in the metadata tables formed according to the method steps of Fig. 2.

[0067] Fig. 12 is a flowchart corresponding to the method step of Fig. 2 corresponding to loading data contained in an extensible document into the relational tables formed in the method steps of Fig. 2.

[0068] Fig. 13 is a flowchart corresponding to the traversal of a node tree defined in the flowchart shown in Fig. 12 for traversing the node tree and inserting data into the relational table schema formed in the method steps of Fig. 2.

[0069] Fig. 14 is an example of a node free discussed with respect to Figs. 12-13 having data and complying with a document-type definition corresponding to Example 1 described in the Background section herein.

DESCRIPTION OF THE PREFERRED EMBODIMENT

[0070] According to the invention, a relational schema is created out of a DTD, metadata is extracted from the DTD that describes the DTD and that illustrates how the DTD maps to the schema, and XML data is loaded into the generated relational schema that adheres to the DTD according to the metadata. In this manner, and as a direct result of the metadata analysis and storage, the data semantics implied by the XML are maintained so that more accurate and efficient management of the data can be performed.

[0071] Starting with a DTD for XML documents containing data (rather than analyzing the relationships between the actual elements of the XML data), all of the information in the DTD is captured into metadata tables, and then the metadata tables are queried to generate the relational schema. Then, the data contained in the XML document can be loaded into the generated relational schema. This method can be described by three broad steps:

[0072] First, the DTD is stored as metadata in tables -- i.e., the metadata is used to describe the information of the DTD associated with the XML documents. This approach provides flexibility to manipulate the DTD by standard SQL queries.

[0073] Second, a relational schema is generated from the metadata stored in the metadata tables. This step provides additional flexibility in that, although the relational schema can be directly mapped from the metadata tables according to the invention, the metadata tables can also be queried to do optimizing or restructuring on the metadata tables representative of the XML data structure stored in the DTD.

[0074] Third, the data contained in the XML document is extracted from the document and stored into the tables defined by the relational schema, which is generated in the previous step, by using the associated metadata tables.

[0075] According to the invention, the inventive metadata-driven approach includes the following beneficial characteristics:

[0076] *For Storing*: All of the information contained in the DTD is captured in the metadata tables. It is anticipated that the XML document and the DTD can be reconstructed from the relational data and metadata as needed.

[0077] *For Mapping*: The generated relational schema is rich in relationships that are useful in processing queries.

[0078] *For Loading*: Mappings between the XML document and the final relational schema are captured in the metadata tables. It is contemplated that the relational data can be synchronized with XML data, which means whenever there is a data update in the relational data, the effect is also reflected in the XML data, and vice versa.

[0079] It will be understood that it has been assumed that there exists only one external DTD file for compliant XML documents and that the file has no nested DTDs, and that there is no internal DTD in the XML files. Of course, to the extent that XML documents are encountered that include these items, this requirement can be achieved by pre-processing XML documents with nested or internal DTDs as needed.

FIGURE 1

[0080] Turning now to the drawings and to Fig. 1 in particular, a system 10 is shown for automatically loading a document 12 into a relational database 14. As can be clearly seen from Fig. 1, the document 12 comprises a first data portion 16 and a second document definition portion 18. It will be understood that the document 12 is preferably an XML

document, the first data portion 16 is preferably a compliant set of tagged data normally found in a document formatted in the XML language, and the tags are compliant with the second document definition portion 18 that comprises a document-type definition (DTD) as is well known in the art (the document 12 is shown surrounded by a broken line in Fig. 1 indicating a possible separated of the DTD 18 and the data 16 since the DTD 18 can be provided separately from the XML data 16 as is also well known). As can also be clearly seen from Fig. 1, the relational database 14 comprises a first data storage portion 20 and a second data definition portion 22. It will be understood that the relational database 14 can be any of the well-known relational databases. The first data storage portion 20 is typically, and preferably, a set of tables in the relational database 14. The second data definition portion 22 preferably comprises a relational schema as is typically used to model, outline or diagram the interrelationship between tables in a relational database.

[0081] It is an important feature of this invention that the DTD 18 (i.e., the second document definition portion 18) is loaded by the system 10 and used in metadata format to generate the relational schema of the second data definition portion 22. Then, the XML data stored in the first data portion 16 of the XML document 12 is loaded by the system 10 into the tables making up the first data storage portion 20.

[0082] In order to accomplish these functions, the system 10 comprises an extractor 24, an optimizer 26, a generator 28 and a loader 30 all of which are interconnected to a storage unit 32. As contemplated by this invention, the storage unit 32 comprises at least a metadata table storage portion 34 and a pattern mapping table storage portion 36.

[0083] According to the method of this invention, which will be hereinafter described in greater detail, the system 10 reads the DTD 18 with the extractor 24 and stores data representative of the DTD 18 in metadata tables in the metadata tables storage portion 34. From the data stored in metadata tables, the generator 28 generates the relational schema 22 in the relational database 14. In an optional loop, the optimizer 26 can massage the data stored in the metadata tables 34 to create a more efficient set of inputs for the generator 28 which, in turn, results in the generation of a more efficient relational schema 22.

[0084] Next, once the relational schema 22 has been generated by the generator 28, a pattern-mapping table 36 is generated from the metadata tables and fed as an input to the loader 30 (in addition to the input of the XML data 16 from the document 12) which, in turn, provides an input to load the tables 20 and the relational database 14 with the XML data 16 stored in the document 12.

FIGURE 1A

[0085] The automatic loading of an XML document 12 into a relational database 14 according to a document-type definition 18 contained in the XML document 12 is shown in greater detail in Fig. 1A. One feature of this invention is the importation of the information contained in the document-type definition 18 to the extractor 24 to create the metadata tables 34 (referred to herein as DTDM, short for document-type definition metadata). As can be seen from Fig. 1A., the metadata tables 34 comprise a DTDM-Item table 90 generally made up of elements and groups defined in the DTD 18, a DTDM-Attribute table 92 generally made up of Attribute of elements and groups contained in the DTD 18, and a DTDM-Nesting table 94 generally made up of nesting relationships contained in the DTD 18 as identified by the extractor 24.

[0086] The metadata tables 34 are then fed to the generator 28 and, optionally, the optimizer 26, to create the link pattern and pattern mapping tables 36. It should be understood that the optimizer 26 is entirely optional and can be omitted without departing from the scope of this invention. When used, the optimizer 26 provides the additional benefits discussed herein. The tables 36 comprise an IM-Item table 96 which contains mapping information relating to the DTDM-Item table 90, an IM-Attribute table 98 that contains mapping information relating to the DTDM-Attribute table 92, an IM-Nesting table 100 that contains mapping information relating to the DTDM-Nesting table 94, and a TS-JC table 102 for containing table schema and join constraint information for assisting in the generation of the table schema 22 for the relational database 14. It will be understood that the metadata tables 34 are preferably necessary for generation of the table schema 22. However, it has been found that the generation of the link pattern and pattern mapping tables 36 can result in the generation of a more efficient table schema 22 for the

relational database 14. Then, either the metadata tables 34 or, if the optimizer 26 is employed, the pattern and pattern mapping tables 36 are fed to the loader 30 to create and fill the tables 20 and the relational database 14 according to the generated table schema 22 therein.

FIGURE 1B

[0087] A schematic representation of the tabular interaction according to this invention is shown in Fig. 1B wherein the tables shown in Fig. 1B refer to the components of the invention shown in Figs. 1 and 1A with like reference numerals. Dashed lines interconnecting the illustrated tables indicate a relationship between a field of one table with a field in the connected table.

[0088] Turning now to Figs. 2-13, the method of automatically loading XML data into a relational database according to a relational schema defined by a document-type definition will now be described.

FIGURE 2

[0089] Fig. 2 describes broad method steps of the invention, shown mainly by the steps surrounded by a double-line frame, wherein step 40 indicates that data representative of the DTD 18 is stored (via the extractor 24) into the metadata tables storage portion as metadata representative of the DTD 18 which is referred to as the DTDM tables 34.

[0090] Step 42 indicates the second broad method step of this invention wherein the relational schema 22 is generated (via the generator 28) from the DTDM tables.

[0091] Step 44 indicates the final broad method step of this invention wherein the XML data 16 from the document 12 is loaded (via the loader 30) into the tables 20 of the relational database 14 according to the relational schema 22 generated in step 42.

[0092] Fig. 2 also describes more detailed method steps for each of the broad method steps 40, 42 and 44.

[0093] Namely, step 40 of storing the DTD 18 into the DTDM tables 90, 92 and 94 preferably comprises steps of creating and filling the DTDM-Item table 90 in the metadata tables 34 (shown by reference number 46 and described in greater detail in Fig. 3), creating and filling a DTDM-Attribute table 92 in the metadata tables 34 (shown by

reference number 48 and described in greater detail in Fig. 4), creating and storing a DTDM-Nesting table 94 in the metadata tables 34 (shown by reference number 50 and described in greater detail in Fig. 5), and initializing a pattern mapping table (shown by reference number 58 and described in greater detail in Figs. 9-11).

[0094] Step 42 of creating the relational table schema 22 from the metadata tables 34 preferably comprises the steps of creating tables in the relational database 14 (shown by reference number 52 and described in greater detail in Fig. 6), adding columns to the tables created in step 52 to correspond to attributes from the metadata tables 34 created in step 48 (shown by reference number 54 and described in greater detail in Fig. 7), and adding nesting relationships indicated by the DTDM-Nesting table 94 stored in the metadata tables 34 in step 50 (shown by reference number 56 and described in greater detail in Fig. 8).

[0095] Step 44 of loading the XML data 16 of the document 12 into the tables 20 of the relational database 14 according to the relational schema 22 generated in step 42 preferably comprises the step of loading the XML data 16 contained in the document 12 into the tables 20 of the relational database 14 according to the relational schema 22 generated herein (shown by reference number 60 and described in greater detail in Figs. 12-13).

[0096] It will be understood that the focus of the metadata extraction steps begins with three empty DTDM tables 34 -- the DTDM-Item table 90, the DTDM-Attribute table 92, and the DTDM-Nesting table 94, the function and features of which will be explained in detail below.

[0097] The DTD 18 is first stored into the metadata tables 34 so that it can be optionally restructured, and then the relational schema 22 can be generated from the metadata tables 34. The storing stage identifies the characteristics of the DTD 18, and stores it as the metadata tables 34. The (optional) restructuring stage can identify the multi-valued attributes of the DTD 18, and can also identify items that could be represented as attributes. Mapping the DTD 18 into the relational schema 22 is achieved by applying mapping rules defining transformations over the metadata tables 34 storing the DTD 18.

[0098] One initial step is identifying the types of the objects that will be found in these types of data-containing XML documents 12. Three kinds of metadata have been identified as being relevant for storing these properties: items, attributes, and relationships. The three metadata tables 34 for storing the items, attributes and relationships defined in the DTD 18 and the properties of the DTD 18 captured by each table 90, 92 and 94 are defined as will be hereinafter described. Every item, attribute and nesting relationship in the tables 90, 92 and 94 is preferably assigned a unique `ID` as will be described in greater detail below.

[0099] An item represents an object in the DTD 18 that contains, or is contained by, other objects. An attribute is a property of an item. An item can have multiple unique attributes. The attributes of an element in a DTD are all the attributes of the corresponding item in the metadata generated by the invention described herein. A nesting relationship is used to show the hierarchical relationships between two items. It denotes that a child item is directly nested in a parent item.

FIGURE 3

[00100] The step of creating and filling the DTDM-Item table 90 identified by reference number 46 in Fig. 2 will now be described in greater detail with respect to Fig. 3. As illustrated in Fig. 3, processing moves to step 62 in which, initially, a pair of default items are created in the DTDM-Item table 90 referred to as "`PCDATA`" and "`GROUP`" (also referred to as `ANY_GROUP` in the figures). Proposed SQL statements that could accomplish this task are shown in the note associated with step 62 referred to by reference number 64. Once these initial default items are created in step 62, processing moves to step 66 which initiates a loop for each Element type declaration in the DTD 18. For each of the Element type declarations, a row is created in the DTDM-Item table 90 as shown by the proposed SQL statement in note 68. Processing then moves to decision block 70 in which it is determined whether any additional Element type declarations exist in the DTD 18. If so, processing returns to step 66. If not, processing ends.

[00101] The DTDM-Item table 90 preferably stores element types and groups. The table 90 captures the existence property of an element as the Type of the item. It also captures a grouping property by creating a new group item for each group.

DTDM-Item Table Field Contents	
Fields	Meaning
ID	Internal ID for Items.
Name	Element Type or Group Name.
Type	Defines the type of this item within this domain: PCDATA, ELEMENT.ELEMENT, EMPTY, ELEMENT.ANY, ELEMENT.MIX, and GROUP

[00102] The Type field defines the type of an item, i.e., type of the element content in an element type declaration. **ELEMENT.ELEMENT** means an element content. **ELEMENT.MIX** means a mix content. **ELEMENT.EMPTY** means an **EMPTY** content. **ELEMENT.ANY** means an **ANY** content. There are two new item types, i.e., **PCDATA** and **GROUP.PCDATA,** means a **PCDATA** definition, and **GROUP** means a group definition.

FIGURE 4

[00103] The step of creating and filling the DTDM-Attribute table 92 identified by reference number 48 in Fig. 2 will now be described in greater detail with respect to Fig. 4. As shown in Fig. 4, processing moves to step 72 in which the **Item_ID** of **PCDATA** is retrieved from the DTDM-Item table 90. Processing then moves to step 74 in which default attribute values for **PCDATA** are created in the DTDM-Attribute table 92. A proposed SQL statement for accomplishing this task is provided in note 76 associated with step 74. Processing then moves to step 78, which initiates a loop for each element type declaration in the DTD 18.

[00104] Processing then moves to step 80 in which the **Item_ID** for each of the element type declarations in the DTDM-Item table 90 is retrieved. Processing then moves to step 82, which initiates a loop for each attribute of this particular element type. For each attribute of this element type declaration, a row is inserted into the DTDM-Attribute table 92 providing attribute information corresponding to the elements of the DTDM-Item table 90. A proposed SQL statement for accomplishing this task is provided in note 84 associated with step 82.

[00105] After each row insertion into the DTDM-Attribute table 92, processing moves to decision block 86 to determine whether additional attributes exist for this element type. If so, processing returns to step 82 to process additional attributes. If not, processing moves to decision block 88 to determine whether additional elements exist in the DTD 18 (i.e., as stored in the DTDM-Item table 90). If so, processing returns to step 78. If not processing ends.

[00106] The DTDM-Attribute table 92 stores attributes of elements or groups.

DTDM-Attribute Table Field Contents	
Fields	Meaning
ID	Internal ID of this attribute.
PID	ID of parent items of this attribute.
Name	Name of this attribute, e.g., AuthorIDs, id.
Type	Type of the attribute, e.g., ID and IDREFS.
Default	A keyword or a default literal value of this attribute, e.g., #IMPLIED.

[00107] Note that, for now, the **ID/IDREF**(s) attributes that represent the element reference properties are stored simply as attributes. Later, during a mapping stage, the element reference property will be captured and stored in an additional metadata table, denoted as the TC-JS table 102 in Fig. 1A (and JoinConstraint 102 in Fig. 1B).

FIGURE 5

[00108] The step of creating and storing the DTDM-Nesting table 94 identified by reference number 50 in Fig. 2 will now be described in greater detail with respect to Fig. 5. Processing moves to step 402 in which the DTDM-Nesting table 94 is initialized. Pseudopodia indicative of the steps performed in step 402 are given in note 404 associated with step 402. Processing then moves to step 406, which initiates a loop for every element type declaration found in the DTD 18. Processing then moves to step 408, which retrieves the type, i.e., **MIXED**, **PCDATA**, **ANY**, **ELEMENT**, and **EMPTY** of the particular element type being examined in the loop initiated at step 406. Processing then moves to step 410 in which the DTDM-Item table 90 is queried to return the identification (**ID**) of the element type identified in step 408.

[00109] Processing then moves to decision block 412 which determines whether the current element type is of type **MIXED**. If so, processing moves to step 414 in which a new group is created in the DTDM-Item table 90 with type **CHOICE**, wherein a proposed SQL statement to accomplish this task is shown in note 416 associated with step 414. Processing then moves to step 418 in which a nesting relationship is created from the current element type to the newly-created group as shown by the proposed SQL statement in note 420 associated with step 418. Processing then moves to step 422 in which a nesting relationship is created from this group to element type **PCDATA**. Processing then moves to step 424 in which all of the nesting relationships from this group to its children are created by function fill_ DTDM_Nesting_Item shown in detail in Fig. 5A (via indicator 5A referred to by numeral 444). Processing then moves to decision block 426 in which it is determined whether there are additional element types in the DTD 18 to be processed for the loop initiated at step 406. If so, processing returns to step 406.

[00110] If the test performed at decision block 412 fails, processing moves to decision block 428 which determines whether the current element type is of type **ANY**. If so, processing moves to step 430 in which one relationship from the current element is created to the item titled **ANY_GROUP**. A proposed SQL statement to accomplish this task is identified in note 432 associated with step 430 in Fig. 5. Processing then moves to decision block 426, which has been previously described.

[00111] If the test performed at decision block 428 fails, processing moves to decision block 434 which determines whether the current element type is of type **PCDATA**. If so, processing moves to step 436 in which a relationship to the previously-created **PCDATA** item is created in accordance with the proposed SQL statement shown in note 438 associated with step 436 in Fig. 5. Processing then moves to decision block 426, which has been previously described.

[00112] If the test performed at decision block 434 fails, processing moves to decision block 440 which determines whether the current element type is of type **ELEMENT**. If so, processing moves to step 442 which calls a function titled fill_DTDM_Nesting_Item (element_type,-1), the details of which are described in Fig.

5A (by the connector identified with 5A and indicated by numeral 444). After this function has completed, processing moves to decision block 426, which has been previously described.

[00113] If the test performed at decision block 440 fails, processing moves to step 446 which notes that the current element type is **EMPTY**. Processing then moves to decision block 426, which has been previously described.

[00114] Once all of the elements have been processed, i.e., all of the element type declarations as identified in the loop initiated at step 406 have been processed, processing ends.

FIGURE 5A

[00115] The contents of the function (i.e., fill_DTDM_Nesting_Item) identified by the "5A" connectors 444 of Fig. 5 are described in greater detail with respect to Fig. 5A. Once this function is called, processing moves to decision block 446 in which it is determined whether the nesting relationships of the parent item are to be copied into the group. If so, processing moves directly to step 448 in which the **group_ID** is treated as an **element_ID**. If not, processing moves to step 450 in which the DTDM-Item table 90 is queried to determine the **ID** of the parent item (i.e., element type or group) as **element_ID**. A proposed SQL statement to accomplish this task is shown in note 452 associated with step 450 in Fig. 5A. In either case, processing then moves to step 454, which initiates a loop for the group or element identified in step 448 as **element_ID**. Processing then moves to step 456, which retrieves the object reference to the current element type or group being processed. Processing then moves to decision block 458, which determines whether the element type or group corresponding to the object reference identified in step 456 is of type **GROUP**. If so, processing moves to step 460, which retrieves the group **ID** and stores this **ID** in variable **ref_ID**. If not, processing moves to step 462 in which the type is determined to be an element type declaration. Following either step 460 or step 462, processing then moves to step 464 in which the previously-determined object reference is stored into the DTDM-Nesting table 92 in a manner consistent with note 466 which shows a proposed SQL statement for

accomplishing this task. Processing then moves to decision block 466 in which it is determined whether any additional references need to be processed. If so, processing returns to step 454. If not, processing terminates and returns to continue processing at the point Fig. 5 was left via connector 444.

[00116] The DTDM-Nesting table 94 captures relationships between different items, i.e., nesting, schema ordering, and occurrence properties.

DTDM-Nesting Table Field Contents	
Fields	**Meaning**
ID	Internal ID of this nesting relationship.
FromID	ID of parent item of this nesting relationship.
ToID	ID of child item of this nesting relationship.
Ratio	Cardinality between the parent element and child element.
Optional	Used to indicate whether a child element. True if existence of the child is optional, (i.e., *, ?). False otherwise.
Index	The schema order of the child element.

[00117] Fields `FromID` and `ToID` reference a parent item and a child item that participate in a nesting relationship. The `Index` field captures the schema ordering property; it denotes the position of this child item in the parent item's definition. In a sequence group, each child item will have a different value for indices (i.e., 1, 2, ...); for all children in a choice group, the index fields will all be the same (i.e., 0).

[00118] The occurrence property for a child element is captured by a combination of the `Ratio` and `Optional` fields. The `Ratio` field shows cardinality between the instances of the parent item and of the child item. Note that, since the nesting relationships are always from one element type to its sub-elements in the DTD 18, there are only one-to-one or one-to-many nesting relationships in the `Ratio` column. The many-to-one and many-to-many relationships are not captured by the `Ratio` field but rather are captured by `ID/IDREF(S)` attributes in the DTD. The `Optional` field has value of `true` or `false` depending on whether this relationship is defined as optional in the DTD. The

following table shows how the `Ratio` and `Optional` fields combine to represent the occurrence properties:

Occurrence Property Indicators		
Occurrence Property	Ratio	Optional
No Indicator	1:1	false
?	1:1	true
+	1:n	false
*	1:n	true

[00119] The steps for extracting the metadata tables 90, 92 and 94 can be summarized as follows:

[00120] **Create one `PCDATA` Item.** This one item will represent all occurrences of `#PCDATA` in the DTD. The `PCDATA` item will be used to convert all element-to-`PCDATA` relationships, such as found in a mixed content definition, to element-to-element relationships, thus unifying these two types of nesting relationships. A `PCDATA` item has one attribute called `value` that is used to capture the text value of this `PCDATA` item.

[00121] **Create an item for each element type declaration.** Tuples in the DTDM-Item table 90 are the elements directly defined by the DTD 18.

[00122] **Create an item for each grouping relationship in each element type declaration.** For each element type declaration in a DTD 18, a group item is created for each group in the item, and the group in the element type declaration is replaced with the corresponding group item. In Example 1, items 14 to 16 represent the groups (`author* | editor`), (`author, affiliation?`), and (`book | monograph`) respectively. Defining each group as an item is used to convert nested groups into nesting relationships between items.

[00123] For example, the definition of element `book` shows that `book` is composed of `booktitle`, and a group of authors or an editor. A new item `G1` would thereby be defined for the group (`author* | editor`), the element definition of `book` would be changed to `<!ELEMENT book (booktitle, G1)>`.

[00124] **Store nesting relationships.** After defining the `PCDATA` item and all group items, the hierarchical definitions of elements can be described as nesting relationships

between two items. An element definition is a sequence (or choice) of *n* sub-elements stored as *n* nesting relationships with index fields in the DTDM-Nesting table 94.

[00125] For example, the element definition `<!ELEMENT book (booktitle, G1)>` has two nesting relationships, i.e., between items `book` and `booktitle`, and between items `book` and `G1`. These nesting relationships are shown in the DTDM-Nesting table 94 constructed in accordance with the DTD 18 of Example 1 (with `ID`s 1 and 2).

[00126] **Store attributes**. For items with attributes defined, those attributes are stored in the DTDM-Attribute table 92. For example, the attribute `AuthorIDs` of item `Contactauthors` is stored in DTDM-Attribute table 92 (with `ID` 1) constructed in accordance with Example 1.

[00127] **Store the `ANY` element type definitions**. An `ANY`-typed element can contain a mix of `PCDATA` and any defined element types, and thus can have relationships with all other element types. To capture that relationship, a choice group item is created called `ANY GROUP` (`AG`). Every element type definition with content `ANY` expresses its relationships with all other element types with a one-to-many nesting relationship with the `AG` item. Using Example 1, row 17 in the DTDM-Item table 90 is the `AG` group, and nesting relationships with `ID` 24 through 36 are between this `AG` group and each of the other element items and the `PCDATA` item, i.e., between this `AG` group and items 1 through 13, respecitvely, in the example described herein conforming to the DTD 18 of Example 1. The `Affiliation` item, which is an `ANY` element type declaration, has a one-to-many nesting relationship to the `AG` group (see line 23 in the DTDM-Nesting table 94 below).

[00128] Once the metadata has been extracted and mapped from the DTD 18 shown in Example 1 in the Background section, the metadata tables 34 have the following structure:

[00129] The DTDM-Item table 90:

ID	Name	Type
1	PCDATA	PCDATA
2	book	ELEMENT.ELEMENT
3	booktitle	ELEMENT.MIX
4	article	ELEMENT.ELEMENT
5	title	ELEMENT.MIX
6	contactauthors	ELEMENT.EMPTY
7	monograph	ELEMENT.ELEMENT
8	editor	ELEMENT.ELEMENT
9	author	ELEMENT.ELEMENT
10	name	ELEMENT.ELEMENT
11	firstname	ELEMENT.MIX
12	lastname	ELEMENT.MIX
13	affiliation	ELEMENT.ANY
14	G1	GROUP
15	G2	GROUP
16	G3	GROUP
17	AG	GROUP

[00130] The DTDM-Attribute table 92:

ID	PID	Name	Type	Default
1	6	authorIDs	IDREFS	#REQUIRED
2	8	name	CDATA	#REQUIRED
3	9	id	ID	#REQUIRED
4	1	value	PCDATA	#REQUIRED

[00131] The DTDM-Nesting table 94:

ID	FromID	ToID	Ratio	Optional	Index
1	2	3	1:1	false	1
2	2	14	1:1	false	2
3	14	9	1:n	true	0
4	14	8	1:1	false	0
5	3	1	1:1	false	0
6	4	5	1:1	false	1
7	4	15	1:n	false	2
8	15	9	1:1	false	1
9	15	13	1:1	true	2
10	4	6	1:1	true	3
11	5	1	1:1	false	0
12	7	5	1:1	false	1
13	7	9	1:1	false	2
14	7	8	1:1	false	3
15	8	16	1:n	true	1
16	16	2	1:1	false	0
17	16	7	1:1	false	0
18	9	10	1:1	false	1
19	10	11	1:1	true	1
20	10	12	1:1	false	2
21	11	1	1:1	false	0
22	12	1	1:1	false	0
23	13	17	1:n	true	1
24	17	1	1:1	false	0
25	17	2	1:1	false	0
26	17	3	1:1	false	0
27	17	4	1:1	false	0
28	17	5	1:1	false	0
29	17	6	1:1	false	0
30	17	7	1:1	false	0
31	17	8	1:1	false	0
32	17	9	1:1	false	0
33	17	10	1:1	false	0
34	17	11	1:1	false	0
35	17	12	1:1	false	0
36	17	13	1:1	false	0

[00132] Any discussion based on examples (and, specifically, Example 1) refer to the above metadata table examples.

[00133] The rules used for mapping the DTD 18 are described (as stored as described above), into the relational schema 22 are described in the following. The basic

idea behind these rules is that each item is to be mapped into a relational table 20 in the database 14 according to generated schema 22.

[00134] As discussed previously, the elements in XML documents are ordered. This order is retained in the newly-generated relational table schema. Groups are not directly shown in the XML document, so they do not have an order property.

FIGURE 6

[00135] The step of creating tables 20 in the relational database 14 (identified by reference number 52 in Fig. 2) will now be described in greater detail with respect to Fig. 6. Turning to Fig. 6, processing moves to step 120 in which a query of the DTDM-Item table 90 is performed to return all of the item types stored in the DTDM-Item 90 table. A proposed SQL statement to accomplish this task is shown in note 122 associated with step 120 in Fig. 6. Processing then moves to step 124, which initiates a loop for every item returned in the recordset selected in step 120. Processing within the loop then moves to step 126 wherein a table 20 is created in the relational database 14 with some key-type default fields, wherein the table name created in the database 14 corresponds to the **Name** field in the DTDM-Item table 90 as returned in the recordset in step 120. A proposed SQL statement to accomplish this task is shown in note 128 associated with step 126 in Fig. 6. Processing then moves to decision block 130, which determines whether additional items exist for processing. If so, processing returns to step 124. If not, processing ends.

[00136] These steps perform a first mapping on the DTDM-Item table 90. That is, for each **ELEMENT.*** and **PCDATA**-typed item defined in the DTDM-Item table 90, a table is created with two default columns: "**iid**" and "**order**". For each **GROUP**-typed item, a table is created with only an "**iid**" column.

[00137] The metadata tables 34 are queried to get all non-group items by:

```
SELECT name, type FROM DTDM-Item WHERE type LIKE "ELEMENT.%" OR type =
"PCDATA"
```

[00138] After the name and type of the item are retrieved, the following queries are performed to create the tables from the query recordset returned. So, for the type of "`ELEMENT`.*" and "`PCDATA`" item in the query recordset result, the following query is issued to create a table for each of them, (e.g., an item called `ItemTable` with a default primary key "`iid`" and a column called "`order`" in an `INTEGER` format):

```
CREATE TABLE ItemTable (iid INTEGER, order INTEGER, PRIMARY KEY iid)
```

[00139] For items of other types, a table will be created in the form such as:

```
CREATE TABLE ItemTable (iid INTEGER, PRIMARY KEY iid)
```

[00140] After identifying the basic tables and their required columns, any other columns that are appropriate for those tables must be determined. First, columns are added, which represent the attributes of its parent item, to each table corresponding to that item. Second, the columns of the various tables are interconnected according to the detected nesting relationships therebetween.

FIGURE 7

[00141] The step of adding columns in the tables created in step 52 for attributes in the DTDM-Attribute table 92 (as identified by reference number 54 in Fig. 2) will now be described in greater detail with respect to Fig. 7. As shown in Fig. 7, processing moves to step 132 in which a join query of the DTDM-Attribute table 92 with the DTDM-Item table 90 is performed to return all of the attributes of an item from data contained in the DTDM-Attribute table 92 and the DTDM-Item tables 90. A proposed SQL statement to accomplish this task is shown in note 134 associated with step 132 in Fig. 7.

[00142] Processing then moves to step 136, which initiates a loop for every Attribute returned in the recordset selected in step 132. Processing then moves to step 138 in which, based upon the attribute type of the particular row in the recordset returned in step 132, a column-type variable is determined. A list of applicable column types is shown in note 140 associated with step 138 in Fig. 7.

[00143] Processing then moves to step 142 in which the relational database schema 22 is altered to add this attribute and its type to its parent table. A proposed SQL statement to accomplish this task is shown in note 144 associated with step 142 in Fig. 7. Processing then moves to decision block 146 to determine whether additional attributes need to be processed. If so, processing returns to step 136. If not, processing ends.

[00144] In this part of the inventive method herein, these steps perform a second mapping on the DTDM-Attribute table 92. For each tuple in the DTDM-Attribute table 92, a column is created, named with the "Name" property of the tuple in the relational table that is identified by the **pid** index of the tuple. All of the column domains are preferably strings, since the DTD 18 preferably only has one data type, i.e., **CDATA**, **PCDATA**. In this invention, it is not necessary to perform additional parsing on the data values to determine their data types, although doing so would not depart from the scope of this invention.

[00145] More generally and by way of summary, the above described steps illustrate that the metadata tables 34 are queried to get all attributes for a given item name **x** (see step 132 and its associated note 134):

```
SELECT A.name, A.type FROM DTDM-Item I, DTDM-Attribute A WHERE I.name =
X AND I.id = A.pid
```

[00146] Then, those attributes returned in the above recordset are placed in the definition of the tables created in the first mapping by issuing the following queries (see step 142 and its associated note 144):

```
ALTER TABLE ItemTable ADD (A1.name A1.type, A2.name A2.type,...)
```

[00147] Here, the A_1, A_2, etc. names and types are the tuples selected from the query issued in connection with step 132 (as described in note 134).

FIGURE 8

[00148] The step of determining and adding nesting relationships to the relational schema 22 (identified by reference number 56 in Fig. 2) will now be described in greater detail with respect to Fig. 8. Turning to Fig. 8, processing moves to step 148 in which a query of the DTDM-Nesting table 94 is performed to return all of the nesting relationships of an item from the data contained in the DTDM-Nesting table 94 and the DTDM-Item table 90 (i.e., to extract all of the item IDs involved in each nesting relationship). A proposed SQL statement to accomplish this task is shown in note 150 associated with step 148 in Fig. 8.

[00149] Processing then moves to step 152 which initiates a loop for every nesting relationship returned in the recordset selected in step 148. Processing then moves to decision block 154 which determines whether the ratio of elements in the particular nesting relationship is a 1-to-1 or a 1 to n relationship. If the decision block 154 determines that the ratio is 1-to-1, processing moves to step 156 in which a foreign key is inserted in the parent table of the relationship. A proposed SQL statement to accomplish this task is shown in note 158 associated with step 156 in Fig. 8. If the decision block 154 determines that the ratio is 1 to n, processing moves to step 160 in which a foreign key is inserted into the child table in the relationship. A proposed SQL statement to accomplish this task is shown in note 162 associated with step 160 in Fig. 8. In either case, processing then moves to re-connector 164 and then to decision block 166 which determines whether additional nesting relationships need to be processed. If so, processing returns to step 152. If not, processing ends.

[00150] Again, in a general summary, a third mapping is thereby performed on the metadata tables 34 in connection with the process shown in Fig. 8. For each tuple `R` in the DTDM-Nesting table 94, the table corresponding to the `from` item is labeled as `S` and the table corresponding to the `to` item as `T` participating in `R`. Then if `R` is a one-to-one nesting relationship, the `iid` of `T` is stored as a foreign key in `S` (i.e., store `T_iid` as a column in `S`); if `R` is a one-to-many nesting relationship, the `iid` of the item `S` is stored as

a foreign key, named `P_T_ iid` (`P` means parent, so it can be thought of as a reverse link), in `T`.

[00151] If the Optional field of this relationship `R` in the DTDM-Nesting table 94 is `false`, then a `NOT NULL` constraint is added on the definition of the table.

[00152] If there is more than one relationship between the two items `S` and `T`, then indices are placed after each column name, e.g., `_T_iid_1`.

[00153] The metadata tables 34 are queried to get pairs of items of all of the nesting relationships (see, e.g., note 150 associated with step 148 in Fig. 8):

```
SELECT F.name, T.name FROM DTDM-Item F, DTDM-Item T, DTDM-Nesting WHERE
F.id=N.fromid AND T.id = N.toid.
```

[00154] In accordance with the invention, this relationship mapping depends upon the ratio inherent to the nesting relationship, so that the corresponding table is updated in the following manner:

[00155] If a one-to-one relationship is detected at decision block 154:

```
ALTER TABLE FromItem ADD (ToItem_iid)
```

[00156] If a one-to-many relationship is detected at decision block 154:

```
ALTER TABLE ToItem ADD (parent_FromItem_iid)
```

[00157] In this approach, many-to-many relationships between different elements are captured by joins on attributes of type `ID` and `IDREF`(s). Different combinations of the `ID`/`IDREF`(s) attribute represent different cardinalities between two elements. The following table represents the possible relationships between two elements depending upon their `ID`/`IDREF`(s) attributes (e.g., of note, the `contactauthors` and `author` elements in Example 1 listed in the Background section have a many-to-many relationship):

		x		
	x:y	ID	IDREF	IDREFS*
	ID	n/a	n:1	n:m
y	IDREF	1:n	n/a	n/a
	IDREFS*	m:n	n/a	n/a

*Or multiple IDREF type of attributes

[00158] A fourth mapping is performed on the metadata tables 34 for the `ID` type of attributes. For the `ID` type of attributes, those attributes are designated as a key of their parent tables.

[00159] A fifth mapping is performed on the metadata tables 34 for the `IDREF`(s) type of attributes. For the `IDREF`(s) type of attributes, join constraints will be added to show meaningful joins between those tables. Each combination of attributes with type `ID` and type `IDREF`(s) is stored as one tuple in a TS-JC table 102 (short for Table Schema/Join Constraints). The `FromColumn` and `ToColumn` store the `ID` of the attribute of type `IDREF(s)` and of type `ID`, respectively.

[00160] The TS-JC table 102 stores these equi-join conditions representing the element reference properties:

Fields	Meaning
ID	Internal ID of this join condition.
FromColumn	Column Join from.
ToColumn	Column Join to.
Occurrence	Times of Join.

[00161] The metadata tables 34 are queried to retrieve all of the attributes having type `ID` by the query:

```
SELECT I.name, A.name FROM DTDM-Item I, DTDM-Attribute A WHERE I.id =
A.pid AND A.type = "ID"
```

[00162] All of the attributes having type `IDREF(S)` are retrieved by the query:

```
SELECT I.name, A.name FROM DTDM-Item I, DTDM-Attribute A WHERE I.id =
A.pid AND (A.type = "IDREF" OR A.type = "IDREFS")
```

[00163] Then, those retrieved recordset items are placed into the TS-JC table 102. The result after performing these mappings on the metadata tables 34 (specifically as updated per Example 1), is shown in the following tables which show essentially a data dictionary of the relational schema 22 for the relational database 14 and also the TS-JC table 102. The data dictionary lists the table names and the columns in each table.

[00164] The relational database data dictionary would look as follows if the metadata for Example 1 is employed:

Table Name	Required Columns	Data Columns	Relationship Columns
PCDATA	iid, order	Value	
book	iid, order		booktitle_iid, G1_iid
booktitle	iid, order	PCDATA_iid	
article	iid, order		title_iid, contactauthor_iid
title	iid, order	PCDATA_iid	
contactauthors	iid, order	authorsIDs	
monograph	iid, order		title_iid, author_iid, editor_iid
editor	iid, order	name	
author	iid, order	id	parent_G1_iid, name_iid
name	iid, order		firstname_iid, lastname_iid
firstname	iid, order	PCDATA_iid	
lastname	iid, order	PCDATA_iid	
affiliation	iid, order		
G1	iid		editor_iid
G2	iid		parent_article_iid, author_iid, affiliation_iid
G3	iid		parent_editor_iid, book_iid, monograph_iid
AG	iid		Parent_affiliation_iid, PCDATA_iid, book_iid, booktitle_iid, article_iid, title_iid, contactauthors_iid, monograph_iid, editor_iid, author_iid, name_iid, firstname_iid, lastname_iid, affiliation_iid

[00165] The TS-JC table 102 would appear as follows:

ID	FromColumn	ToColumn	Occurrence
1	author.id	contactauthors.authorIDs	

[00166] From the examples shown above, it can be seen that there are several relationships between different tables and also, the **IDREFS** type attribute of the DTD 18, e.g., **authorIDs**, is not fully represented by the schema 22 proposed above.

[00167] Two restructuring techniques (i.e., as part of the optional optimizer 26) can be employed according to the invention on the metadata tables 34 to address these shortcomings. First, multiple-value attributes can be identified. Second, elements that should be attributes of other elements can be identified.

[00168] In the DTD 18 (and, of course, any DTD), some attribute types can contain multiple values, e.g. **IDREFS**, **NMTOKENS**, **ENTITIES**. Such attributes can be analogized to a set rather than an attribute, and it is desired to represent their values as sets. Instead of treating the whole attribute as a unitary string of some undetermined length, the values of that attribute are accessed individually. Hence, it is desired to identify these multiple-value types of attributes and convert them into separate tables to access those values. Of course, the transformation of attribute types other than **IDREFS** would be handled in a similar manner.

[00169] By way of example, assuming that a DTDM-Item **E** has a multiple-value attribute **A**. For each **A**, another item named **E:A** is created. There is only one attribute in this item named **value**. A one-to-many relationship is then created between the item **E** and the item **E:A**. The attribute **A** is then removed from the attribute list of the item **E**. The following paragraphs describe this type of mapping in detail.

[00170] It can be seen from the above metadata expression of Example 1 that the attribute **authorIDs** of item **contactauthors** is of type **IDREFS**. So, a new item **contactauthors_authorIDs** is created and the attribute **authorIDs** with type **IDREFS** is changed into the attribute value of item **contactauthors_authorIDs** with type **IDREF**. This allows the expression of multiple values of attribute **authorIDs** by a new table.

[00171] In a DTD, there are one-to-one relationships between different elements. If an element of type A contains only an `EMPTY` content element, then a later element of type B can be considered to act as a complex attribute of an earlier one. Hence, a technique is proposed to convert these kinds of "complex attribute"-elements into a real attribute which is referred to herein as an inline attribute process.

[00172] Inlining an attribute means, if item `A` to item `B` has a `1:1` nesting relationship and item `B` has no child item, then all of the attributes of item `B` can be inlined into item `A`. Then, the attribute `X` of item `B` is inlined into `A` as `B_X`. This inline technique cannot be applied to one-to-many relationships, because multiple occurrences of the attributes could exist.

[00173] After inlining the attributes by the process discussed above, the number of nesting relationships is consequently reduced, hence the table schema 22 is simpler. It can be appreciated that a group item could also contain other items, so, after inlining attributes, the group item could have attributes which are converted from its children items.

[00174] The inlining process includes multiple iterations, until no additional items can be inlined. In terms of operations on the metadata tables 34, starting from the `PCDATA` element (leaves), items in the DTDM-Nesting table 94 are searched for that have never appeared in the "`FromID`" field, and only appeared in "`ToID`" field with "`Ratio`" as "`1:1`".

[00175] In order to apply the inline operation, the item `B` has to have no child items, and the relationship between item `A` and item `B` must be one-to-one, as follows:

```
SELECT ID
FROM Nesting N
WHERE ToID IN
      (SELECT UNIQUE  ToID as PID,
      FROM Nesting
      EXCEPT
      SELECT UNIQUE FromID as PID
      FROM Nesting) AND
      NOT EXIST
      (SELECT *
      FROM Nesting
      WHERE ToID = N.ToID AND Ratio = "1:n");
```

[00176] For example, using the metadata tables generated in response to Example 1, there are two proposed iterations. During the first iteration, only item `1` (`PCDATA`) is returned. During the second iteration, item `3` (`booktitle`), `5` (`title`), `11` (`firstname`) and `12` (`lastname`) are returned. Then, there no more items satisfy the above conditions.

[00177] After inlining the attributes, the `GROUP`, or `ATTRIBUTE` typed items can be removed which, of course, have no relationship with other items, from the DTDM-Item table 90. However, the items of type `ELEMENT.*` or `PCDATA` cannot be removed, because the elements and `PCDATA` are required during the database data loading phase discussed hereafter in detail. After loading the data from the document 12, those tables can be removed to reduce the degree of redundancy of data.

[00178] There is an additional refinement to make the schema 22 more meaningful. That is, in the table schema 22, all of the attribute names of "`*.PCDATA`.value" are simplified into "`*`". Then, a query can be performed in simpler semantic like:

```
SELECT booktitle FROM book
```

rather than

```
SELECT booktitle.PCDATA_value FROM book
```

[00179] The tables following this paragraph show the metadata tables 34 after being restructured as discussed above. Compared to the initial version of the metadata tables 34 above, it can be seen that, the number of nesting relationships in the DTDM-Nesting table 94) has been reduced to half, while the number of attributes has been increased. This results in bigger tables and less joins across the tables, which is beneficial for join query performance, and is easier to understand.

[00180] The restructured DTDM-Item Table 90 (also referred to herein as the IM-Item table 96):

ID	Name	Type
1	PCDATA	PCDATA
2	book	ELEMENT.ELEMENT
3	booktitle	ELEMENT.MIX
4	article	ELEMENT.ELEMENT
5	title	ELEMENT.MIX
6	contactauthors	ELEMENT.EMPTY
7	monograph	ELEMENT.ELEMENT
8	editor	ELEMENT.ELEMENT
9	author	ELEMENT.ELEMENT
10	name	ELEMENT.ELEMENT
11	firstname	ELEMENT.MIX
12	lastname	ELEMENT.MIX
13	affiliation	ELEMENT.ANY
14	G1	GROUP
15	G2	GROUP
16	G3	GROUP
17	AG	GROUP
18	contactauthors_ authorsIDs	ATTRIBUTE

[00181] The restructured DTDM-Attribute table 92 (also referred to herein as the IM-Attribute table 98):

ID	PID	Name	Type	Default
1	18	value	IDREF	#REQUIRED
2	8	name	CDATA	#REQUIRED
3	9	id	ID	#REQUIRED
4	1	value	PCDATA	#REQUIRED
5	3	PCDATA_value	PCDATA	#REQUIRED
6	5	PCDATA_value	PCDATA	#REQUIRED
7	11	PCDATA_value	PCDATA	#REQUIRED
8	12	PCDATA_value	PCDATA	#REQUIRED
9	17	PCDATA_value	PCDATA	#REQUIRED
10	2	booktitle	PCDATA	#REQUIRED
11	17	booktitle	PCDATA	#REQUIRED
12	4	title	PCDATA	#REQUIRED
13	7	title	PCDATA	#REQUIRED
14	17	title	PCDATA	#REQUIRED
15	10	firstname	PCDATA	#REQUIRED
16	17	firstname	PCDATA	#REQUIRED
17	10	lastname	PCDATA	#REQUIRED
18	17	lastname	PCDATA	#REQUIRED
19	9	name_firstname	PCDATA	#REQUIRED
20	9	name_lastname	PCDATA	#REQUIRED
21	17	name_firstname	PCDATA	#REQUIRED
22	17	name_lastname	PCDATA	#REQUIRED

[00182] The restructured DTDM-Nesting table 94 (also referred to herein as the IM-Nesting table 100):

ID	FromID	ToID	Ratio	Optional	Index
2	2	14	1:1	false	2
3	14	9	1:n	true	0
4	14	8	1:1	false	0
7	4	15	1:n	false	2
8	15	9	1:1	false	1
9	15	13	1:1	true	2
10	4	6	1:1	true	3
13	7	9	1:1	false	2
14	7	8	1:1	false	3
15	8	16	1:n	true	1
16	16	2	1:1	false	0
17	16	7	1:1	false	0
22	12	1	1:1	false	0
23	13	17	1:n	true	1
25	17	2	1:1	false	0
27	17	4	1:1	false	0
29	17	6	1:1	false	0
30	17	7	1:1	false	0
31	17	8	1:1	false	0
32	17	9	1:1	false	0
36	17	13	1:1	false	0
37	6	18	1:n	false	-1

[00183] The following data dictionary representative of the table schema 22 is thereby generated in accordance with this inventive method:

Table Name	Required Columns	Data Columns	Relationship Columns
PCDATA	iid, order	value	
book	iid, order	booktitle	G1_iid
booktitle	iid, order	PCDATA_value	
article	iid, order	title	contactauthor_iid
title	iid, order	PCDATA.value	
contactauthor	iid, order		
monograph	iid, order	title	author_iid, editor_iid
editor	iid, order	name	
author	iid, order	id, name_firstname, name_lastname	parent_G1_iid
name	iid, order	firstname, lastname	
firstname	iid, order	PCDATA_value	
lastname	iid, order	PCDATA_value	
affiliation	iid, order		
G1	iid		editor_iid
G2	iid		parent_article_iid, author_iid, affiliation_iid
G3	iid		parent-editor_iid, book_iid, monograph_iid
AG	iid	PCTDATA_value, booktitle, title, firstname, lastname, name_firstname, name-lastname	parent_affiliation_iid, book_iid, article_iid, contactauthors_iid, monograph_iid, editor_iid, author_iid, affiliation_iid
contactauthors .authorIDs	idd	value	PARENT_contactauthors_id d

[00184] The TS-JC table 102 follows:

ID	FromColumn	ToColumn	Occurrence
1	author.id	contactauthors.authorIDs.value	

[00185] Now, the details of this inventive metadata-driven method that loads an XML document 12 into the relational schema 22 generated above will be discussed.

[00186] The loading process has two general phases as depicted in Fig. 1. First, the pattern-mapping table is generated that is used to capture the mapping between a DTD 18 and the relational schema 22. This includes generating an initial pattern-mapping table

36, and updating this pattern-mapping table to keep track of the actions during the restructuring of metadata tables 34. Second, XML documents 12 are loaded that comply with the DTD 18 using the pattern-mapping table 36.

[00187] An example of a compliant data portion 16 of the XML document 12 is shown below:

[00188] **Example 2. A valid XML Document complying with the DTD of Example 1.**

```
<!xml version="1.0">
<!DOCTYPE article SYSTEM "book.dtd">
<article>
  <title>XML Relation Mapping</title>
  <author id = "xz>
    <name>
      <firstname>Xin</firstname>
      <lastname>Zhang</lastname>
    </name>
  </author>
  <affiliation>
   Department of Computer Science
   Worcester Polytechnic Institute
   Worchester, MA 01609-2280
  </affiliation>
  <author id = "gm">
    <name>
      <firstname>Gail</firstname>
      <lastname>Mitchell</lastname>
    </name>
  </author>
  <affiliation>
   Verizon Laboratories Incorporated
   40 Sylvan Rd.
   Waltham, MA 02451
  </affiliation>
  <author id = "wl">
    <name>
      <firstname>Wang-chien</firstname>
      <lastname>Lee</lastname>
    </name>
  </author>
  <affiliation>
   Verizon Laboratories Incorporated
   40 Sylvan Rd.
   Waltham, MA 02451
  </affiliation>
 <contactauthors authorIDs="xz gm wl">
</article>
```

FIGURES 12-14

[00189] With reference to the drawings and, specifically, to Figs. 12-14, an XML tree model 300 is proposed herein, followed by the introduction of the concept of patterns detected during the importation of the data, the definition of the loading actions, the description of the generation of a pattern-mapping table, the provision of details of the loading algorithm, and finally by a loading example.

[00190] By way of explanation, it is known to parse XML documents into a document object model (DOM)-complaint XML tree structure. Here, a simplified DOM model is proposed for illustration purposes. This model (of the type shown in Fig. 14) is composed of nodes 302 and edges 304. Every node 302 of the tree corresponds to one element in the XML document 12. An edge 304 corresponds to a nesting relationship between elements. The relationships between elements with `ID/IDREF` typed attributes are not shown in this model.

[00191] Every node 302 preferably has a type and possibly one or more attributes. Every attribute preferably has a name and its corresponding value. The `PCDATA` node has only one attribute, named "`value`". Fig. 14 depicts the XML document 12 defined in Example 2 in terms of this tree model 300. For internal nodes 302, the element type is written above the node followed by any attributes and their values. All leaf nodes 302 have type `PCDATA`, and have their values in the "`value`" attribute below the node 302. Arcs between nodes 302 illustrate nesting relationships between the nodes 302. In addition, every node 302 has its document order on it (contained within it (i.e., the order by which a tree traversal routine encounters a particular node 302).

[00192] In the model of Fig. 14, other known types of nodes, e.g., notation, comments, etc., as considered in the well known DOM model have not been included, because this invention focuses primarily on the elements and their attributes.

[00193] When the XML tree model of Fig. 14 is examined, it can be observed that there are three kinds of objects in the tree -- nodes, links, and attributes. Different types of nodes, links and attributes are stored in different parts of the relational tables 20 generated for the relational database 14 in accordance with the previously generated relational

schema 22. A pattern associated therewith is preferably the type of node, link and/or attribute and are referred to herein as a node pattern, a link pattern, and an attribute pattern.

[00194] The node pattern is identical to one item, of whose type is **ELEMENT.*** or **PCDATA**, in the DTDM-Item table 90. It is represented as its item name in the pattern-mapping table 36. The attribute pattern is identical to one attribute in the DTDM-Attribute table 92 and is represented as its full attribute name (including its associated item name) in the pattern-mapping table 36. The link pattern is used to show all possible links between two types of elements that are permitted by the DTD 18 and is represented in the tables below as two element types with an arrow ("→") in the middle.

[00195] There are two types of nesting relationships captured in the DTDM-Nesting table 94, i.e., the test for determining such is whether the relationship involves a group typed item or not. If a nesting relationship does not involve a group-typed item, then it can be directly mapped into a link between the two items participating in that relationship. For a nesting relationship that involves group typed items, the following steps are proposed to generate the link patterns:

[00196] A temporary table link is created by doing a self-join of the DTDM-Nesting table 94 on the group typed items:

```
CREATE Table link AS
SELECT A.FromID, B. ToID
FROM DTDM-Nesting A, DTDM-Nesting B, DTDM-Item C
WHERE  A.ToID = B. FromID AND A.ToID = C.ID and C.type = 'GROUP'
```

[00197] For example, nesting relationship pairs **(2,3)**, **(2,4)** for **GroupID=14** can be determined from the metadata tables 34 according to the example described previously.

[00198] Further self-joins are performed on the table link until all the **FromID** and **ToID** indices are not group typed items. It is contemplated that as many as $n - 1$ iterations

may need to occur until all possible self-joins are performed (for a maximum of *n* level groups).

[00199] Accordingly, the links in are located in this generated link table are the remainder of the link patterns needed.

[00200] The following tables provide the pattern-mapping table 36 with all of the patterns from the metadata tables 34 generated according to the DTD 18 provided in Example 1 and discussed throughout. By way of clarification, different types of patterns have been separated by a solid line in the following table in the following order: (1) node patterns; (2) attribute patterns; and (3) link patterns.

Pattern
PCDATA
book
booktitle
article
title
contactauthors
monograph
editor
author
name
firstname
lastname
affiliation
contactauthors.authorIDs
editor.name
author.id
PCDATA.value
book→booktitle
book→author •
book→editor •
booktitle→PCDATA
article→title
article→author
article→affiliation
article→contactauthors
title→PCDATA
monograph→title
monograph→author
monograph→editor
editor→book
editor→monograph
author→name

```
name→firstname
name→lastname
firstname→PCDATA
lastname→PCDATA
affiliation→PCDATA
affiliation→book
affiliation→booktitle
affiliation→article
affiliation→title
affiliation→contactauthors
affiliation→monograph
affiliation→editor
affiliation→author
affiliation→name
affiliation→firstname
affiliation→lastname
affiliation→affiliation
```


[00201] Patterns are used to identify the type of nodes, links and attributes. Then, the inventive process contemplates the definition of loading actions as will be described below. These loading actions describe how to fill the data in the pattern mapping tables into relational tables 20 for the relational database 14. These loading actions are summarized below as **create**, **update**, **assign** and **decompose**.

[00202] **create(T)** creates a tuple in table **T** with a new "**iid**" and an optional "**order**" if handling an element.

[00203] **update(T:A)** updates a column **A** of the current tuple in table **T** with the value of the attribute in the XML tree (like the one shown in Fig. 14).

[00204] **assign(T:A; S:B)** assigns a column A in table T with another attribute B in table S.

[00205] **decompose(T:A)** decomposes multiple tuples for a multi-value attribute and stores those values into column **A** in table **T**. For example, if a multi-value attribute has value "**v1 v2**", then it will create two tuples with value "**v1**" and "**v2**" respectively.

[00206] The possible mappings between the pattern detected and the loading actions are described in the following paragraphs.

[00207] When a node is encountered, one new tuple is created in the corresponding table.

[00208] When an attribute is encountered, two possible cases result. First, this attribute can be mapped into a column, e.g., `update(T.A)`, and then the column of the tuple in the specific table is updated. Second, this attribute can be mapped into a table, and then multiple tuples in the specific table can be created for each value in this attribute.

[00209] When a link is encountered, three possible cases result. First, the foreign key in one table can be updated with the key value in another table. Second, if there is a group in this link, then a new tuple is created in the group table as well as the corresponding foreign keys are updated. Third, if the child node is inlined in the parent node, then all of the attributes of the child table are copied into the parent table. The details of how to generate those actions are discussed below.

Pattern	Actions
Node: T	create(T)
Attribute:T.A	update(T.A) •decompose(T.value), assign(T_A.parent.T_iid, T.iid))
Link: A→B	assign(A.iid, B.iid) •(create(G), assign(A.G_iid, G.iid), assign(G.B_iid, B.iid) •(create(G), assign(A.G_iid, G.iid), assign(B.parent_G_iid, G.iid) •(create(G), assign(G.parent_A_iid, A.iid), assign(G.B_iid, B.iid) •(create(G), assign(G.parent_A_iid, A.iid), assign(B.parent_G_iid, G.iid) •assign(A.attribute, B.attribute)

[00210] As described with respect to Fig. 14, an XML document 12 in the DOM model 300 consists of nodes 302 and links 304, but it is desired to load the XML data 16 into the relational tables 20 according to the generated schema 22 of the relational database 14. Therefore, mappings are needed between the XML document 12 and the relational tables 20 of the relational database 14.

[00211] Beneficially, those needed mappings are captured in the pattern-mapping table 36, which capture the mapping between patterns to the loading actions. Because

some of the actions create tuples, and some actions use those tuples, the order in the mapping field of the pattern-mapping table 36 is very important.

FIGURES 9-11

[00212] The pattern-mapping table 36 is generated right after the metadata is loaded, and modified during the restructuring. The generation of the pattern-mapping table 36 during each step of the schema-generation process will now be discussed as indicated by the step of initializing the pattern mapping table 36 identified by reference number 58 in Fig. 2 with reference to Figs. 9-11.

[00213] Turning to Fig. 9, the process of initializing the pattern mapping table as identified in Fig. 2 by reference number 58 can be described by two general steps: first, initializing the link table as shown by reference number 168 and then initializing the pattern mapping table as shown by reference number 170.

[00214] Turning to Fig. 10, the step of initializing the link table identified by reference number 168 in Fig. 9 is described in further detail. Processing moves to step 172 in which a first link table is created (e.g., the initial link table is referred to as `Link0`). A proposed SQL statement to accomplish this task is shown in note 174 associated with step 172 in Fig. 10.

[00215] Processing then moves to step 176 and which a variable to control the number of iterations is initialized, namely, `iteration_number` equals zero. Processing then moves to step 178 in which the number of groups from the link table, i.e., the recordset returned in step 172, is retrieved. A proposed SQL statement to accomplish this task is shown in note 180 associated with step 178 in Fig. 10.

[00216] Processing then moves to decision block 182 in which it is determined whether additional groups (beyond the first) need to be processed. If not, processing ends. If so, processing moves to step 184 in which the iteration control number `iteration_number` is incremented. Processing then moves to step 186 in which additional link tables are created (i.e., `Link1`, `Link2`, `Link3`...) for each of the additional iterations. A proposed SQL statement to accomplish this task is shown in note 188 associated with step 186 in Fig. 10.

[00217] Processing then moves to step 190 in which the number of groups is selected from the next newly-created link table. A proposed SQL statement to accomplish this task is shown in note 192 associated with step 190 in Fig. 10. Processing then moves to decision block 194 to determine whether additional groups need to be processed. If so, processing returns to step 184. If not, processing moves to step 196 in which the most recently-created link table is loaded into the link pattern table which is reflected in the process steps of Fig. 11. After step 196, processing ends.

[00218] Turning to Fig. 11, processing moves to step 198 in which all of the **ELEMENT** and **PCDATA** items from the DTDM-Item tables 90 are returned in the recordset. Processing then moves to step 200 which initiates a loop for each item returned in the recordset generated in step 198. Processing then moves to step 202 in which one tuple corresponding to these items is created in a CreateAction table (see Fig. 1B for the interrelation of the CreateAction table with the rest of the pattern mapping tables 36). Processing then moves to step 204 in which mapping corresponding to the tuple generated in step 202 is inserted into the pattern-mapping table 36. Processing then moves to decision block 206 in which it is determined whether additional items of the recordset returned in step 198 need to be processed. If so, processing returns to step 200. If not, processing moves to step 208 in which a query is performed on the DTDM-Attribute table 92 to return a recordset containing all of the attributes thereof. Processing then moves to step 210 in which a loop is initiated for each attribute returned in the recordset generated in step 208. Processing then moves to step 212 in which a tuple is created in an UpdateAction table (see Fig. 1B for the interrelation of the UpdateAction table with the rest of the pattern mapping tables 36). Processing then moves to step 214 in which mapping corresponding to the tuple generated in step 212 is inserted into the pattern mapping table. Processing then moves to decision block 216 which determines whether additional attributes of the recordset generated in step 208 need to be processed according to the loop 210. If so, processing returns to step 210. If not, processing moves to step 218.

[00219] In step 218, a query is performed on a LinkPattern table (see Fig. 1B for the interrelation of the LinkPattern table with the rest of the pattern mapping tables 36) to return a recordset containing all of the links thereof. Processing then moves to step 220 in which a loop is initiated for each link returned in the recordset generated in step 218. Processing then moves to step 222 in which a query is performed to determine all of the groups involved in this particular link. Processing then moves to step 224 in which a loop is initiated for the name of each group.

[00220] Processing then moves to step 226 in which a query is performed to determine a unique identifying `action_ID`. Once this unique `action_ID` is determined, processing moves to step 228 in which a tuple is created in the CreateAction table with the unique `action_ID`. Processing then moves to step 230 in which the tuple is inserted into the pattern matching table 36. After which, processing moves to decision block 232 in which it is determined whether additional groups need to be processed for the name of each group. If so, processing returns to step 224. If not, processing moves ahead to step 234. At step 234, a query is performed to determine all of the (from, to) pairs for this particular link of the loop initiated at step 220. This can be stored in the form of a recordset. Processing then moves to step 236 which initiates a loop for each pair generated in the recordset developed at step 234. Processing then moves to step 238 in which an assigned action is created corresponding to that (from, to) pair. Processing then moves to 240 in which this assigned action is inserted into the pattern mapping table 36 corresponding to the data developed in step 238.

[00221] Processing then moves to decision block 242 in which it is determined whether additional pairs need to be processed in the loop initiated at step 236. If so, processing returns to step 236. If not, processing moves to decision block 244 in which it is determined whether additional links need to be processed in the loop initiated at step 220. If so, processing returns to step 220. If not, processing ends.

[00222] By way of summary, the pattern mapping tables 36 are initialized by putting the generated pattern and the corresponding actions together. For all of the node patterns, e.g., `T, put action create(T)`. For all of the attribute patterns, e.g., `T.A,`

`put action update(T.A)`. For the link pattern, e.g., A→B, there are different cases. If the link pattern is not related to a group, then based on the mapping rule, if the relationship is one-to-one, then `put assign(A.B_iid, B.iid)`, if the relationship is one-to-many, then `put assign(B.parent _A_iid, A.iid)`. If the link pattern is related to a group G, then put G first, then handle the relationship between A→G and G→B separately as described before.

[00223] There are two types of reorganizations of metadata mentioned herein. One type is intended for identity sets -- the result of which is to convert some attributes into items. The effect of this operation on the pattern-mapping table 36 follows. If an attribute `A` of item `I` is changed into item `I_A` with attribute value, then the corresponding actions for that attribute is preferably changed as follows:

```
decompose(I_A.value),
assign(I.A.parent_I_iid, I.iid)
```

[00224] For example, the `contactauthors.authorIDs` is changed to an item named `contactauthors_authorIDs`. Hence the action for attribute pattern `contactauthors.authorIDs` changes to:

```
decompose(contactauthors.authorIDs.value),
assign(contactauthors_authorIDs.parent contactauthors_iid,
contactauthors.iid)
```

[00225] The second type is for inline attributes, which essentially copies the attribute of one item into another item. Hence, if we assume item `B` is inlined as attributes of item `A`, and `B` has `C1`, ..., `Cn` as attributes then all of the assigned loading actions are queried for in the format of `assign(A.B_iid, B.iid)`, which generates a one-to-one relationship between items `A` and `B`. Then, all of the attributes of item `B` are retrieved from the DTDM-Attribute table 92. Then, for every attribute C_i of Item B, the action is replaced with:

```
assign(A.B.C1, B.C1),
assign(A.B.C2, B.C2),
```

```
...
assign(A.B.Cn, B.Cn ),
```

[00226] For example, if `PCDATA` is inlined, then, for the `booktitle→PCDATA` link pattern, the following actions result:

```
assign(booktitle.PCDATA_value, PCDATA.value)
```

[00227] The schema now needs to be cleaned up by replacing all actions with a `*(.*)+.PCDATA_value` pattern with `*(.*)+.`

[00228] The pattern mapping table 36 generated from the metadata described in the example follows:

Pattern	Actions
PCDATA	create(PCDATA)
book	create(book)
booktitle	create(booktitle)
article	create(article)
title	create(title)
contactauthors	create(contactauthors)
monograph	create(monograph)
editor	create(editor)
author	create(author)
name	create(name)
firstname	create(firstname)
lastname	create(lastname)
affiliation	create(affiliation)
contactauthors.authorIDS	update(contactauthor.authorIDs)
editor.name	update(editor.name)
author.id	update(author.id)
PCDATA.value	update(PCDATA.value)
book→booktitle	assign(book.booktitle_iid, booktitle.iid)
book→author •	create(G1), assign(book.G1_iid, G1.iid), assign(author.parent_G1_iid = G1.iid)
book→editor •	create(G1), assign(book.G1_iid, G1.iid), assign(G1.editor.iid, editor.iid)
booktitle→PCDATA	assign(booktitle.PCDATA_iid, PCDATA.iid)
article→title	assign(article.title_iid, title.iid)
article→author	create(G2), assign(G2.parent_article_iid, article.iid), assign(G2.author_iid, author.iid)
article→affiliation	create(G2), assign(G2.parent_article_iid, article.iid), assign(G2.affiliation_iid, affiliation.iid)
article→contactauthors	assign(article.contactauthors_iid,

	contactauthors.iid)
title→PCDATA	assign(title.PCDATA_iid, PCDATA.iid)
monograph→title	assign(monograph.title_iid, title.iid)
monograph→author	assign(monograph.author_iid, author.iid)
monograph→editor	assign(monograph.editor_iid, editor.iid)
editor→book	create(G3), assign(G3.parent_editor_iid, editor.iid), assign(G3.book_iid, book.iid)
editor→monograph	create(G3), assign(G3.parent_editor_iid, editor.iid) assign(G3.monograph_iid, monograph.iid)
author→name	assign(author.name_iid, name.iid)
name→firstname	assign(name.firstname_iid, firstname.iid)
name→lastname	assign(name.lastname_iid, lastname.iid)
firstname→PCDATA	assign(firstname.PCDATA_iid, PCDATA.iid)
lastname→PCDATA	assign(lastname.PCDATA_iid, PCDATA.iid)
affiliation→PCDATA	create(AG), assign(A.G.parent_affiliation_iid, affiliation.iid), assign(AG.PCDATA_iid, PCDATA.iid)
affiliation→book	create(AG), assign(AG.parent_affiliation_iid, affiliation.iid), assign(AG.book_iid, book.iid)
affiliation→booktitle	create(AG), assign(AG.parent_affiliation_iid, affiliation.iid), assign(AG.booktitle_iid, booktitle.iid)
affiliation→article	create(AG), assign(AG.parent_affiliation_iid, affiliation.iid), assign(AG.article_iid, article.iid)
affiliation→title	create(AG), assign(AG.parent_affiliation_iid, affiliation.iid), assign(AG.title_iid, title.iid)
affiliation→ contactauthors	create(AG), assign(AG.parent_affiliation_iid, affiliation.iid), assign(AG.contactauthors_iid, contactauthors.iid)
affiliation→monograph	create(AG), assign(AG.parent_affiliation_iid, affiliation.iid), assign(AG.monograph_iid, monograph.iid)
affiliation→editor	create(AG), assign(AG.parent_affiliation_iid, affiliation.iid), assign(AG.editor_iid, editor.iid)
affiliation→author	create(AG), assign(AG.parent_affiliation_iid, affiliation.iid), assign(AG.author_iid, author.iid)
affiliation • name	create(AG), assign(AG.parent_affiliation_iid, affiliation.iid), assign(AG.name_iid, name.iid)
affiliation→firstname	create(AG), assign(AG.parent_affiliation_iid, affiliation.iid), assign(AG.firstname_iid, firstname.iid)
affiliation→lastname	create(AG), assign(AG.parent_affiliation_iid, affiliation.iid), assign(AG.lastname_iid, lastname.iid)
affiliation→affiliation	create(AG), assign(AG.parent_affiliation_iid, affiliation.iid), assign(AG.affiliation_iid, affiliation.iid)

[00229] Then, the following pattern-mapping table results:

Pattern	Actions
PCDATA	create(PCDATA)
book	create(book)
booktitle	create(booktitle)
article	create(article)
title	create(title)
contactauthors	create(contactauthors)
monograph	create(monograph)
editor	create(editor)
author	create(author)
name	create(name)
firstname	create(firstname)
lastname	create(lastname)
affiliation	create(affiliation)
contactauthors.authorIDS	decompose(contactauthors_authorIDs.value), assign(contactauthors_authorIDs.parent_. contactauthors_iid, contactauthors.iid)
editor.name	update(editor.name)
author.id	update(author.id)
PCDATA.value	update(PCDATA.value)
book→booktitle	assign(book.booktitle_iid, booktitle.iid)
book→author •	create(G1), assign(book.G1_iid, G1.iid), assign(author.parent_G1_iid = G1.iid)
book→editor •	create(G1), assign(book.G1_iid, G1.iid), assign(G1.editor.iid, editor.iid)
booktitle→PCDATA	assign(booktitle.PCDATA_value, PCDATA.value)
article→title	assign(article.title, title.PDATA_value)
article→author	create(G2), assign(G2.parent_article_iid, article.iid), assign(G2.author_iid, author.iid)
article→affiliation	create(G2), assign(G2.parent_article_iid, article.iid), assign(G2.affiliation_iid, affiliation.iid)
article • contactauthors	assign(article.contactauthors_iid, contactauthors.iid)
title→PCDATA	assign(title.PCDATA_value, PCDATA.value)
monograph→title	assign(monograph.title, title.PCDATA_value)
monograph→author	assign(monograph.author_iid, author.iid)
monograph→editor	assign(monograph.editor_iid, editor.iid)
editor→book	create(G3), assign(G3.parent_editor_iid, editor.iid), assign(G3.book_iid, book.iid)
editor→monograph	create(G3), assign(G3.parent_editor_iid, editor.iid) assign(G3.monograph_iid, monograph.iid)
author→name	assign(author.name_firstname, name.firstname), assign(author.name_lastname, name.lastname)
name→firstname	assign(name.firstname, firstname.PCDATA_value)
name→lastname	assign(name.lastname, lastname.PCDATA_value)
firstname→PCDATA	assign(firstname.PCDATA_value, PCDATA.value)
lastname→PCDATA	assign(lastname.PCDATA_value, PCDATA.value)

affiliation→PCDATA	create(AG), assign(A.G.parent_affiliation_iid, affiliation.iid), assign(AG.PCDATA_iid, PCDATA.iid)
affiliation→book	create(AG), assign(AG.parent affiliation iid, affiliation.iid), assign(AG.book_iid, book.iid)
affiliation→booktitle	create(AG), assign(AG.parent affiliation iid, affiliation.iid), assign(AG.booktitle_iid, booktitle.iid)
affiliation→article	create(AG), assign(AG.parent affiliation iid, affiliation.iid), assign(AG.article_iid, article.iid)
affiliation→title	create(AG), assign(AG.parent affiliation iid, affiliation.iid), assign(AG.title_iid, title.iid)
affiliation→ contactauthors	create(AG), assign(AG.parent affiliation iid, affiliation.iid), assign(AG.contactauthors_iid, contactauthors.iid)
affiliation→monograph	create(AG), assign(AG.parent affiliation iid, affiliation.iid), assign(AG.monograph_iid, monograph.iid)
affiliation→editor	create(AG), assign(AG.parent affiliation iid, affiliation.iid), assign(AG.editor_iid, editor.iid)
affiliation→author	create(AG), assign(AG.parent affiliation iid, affiliation.iid), assign(AG.author_iid, author.iid)
affiliation→name	create(AG), assign(AG.parent affiliation iid, affiliation.iid), assign(AG.name_firstname, name.firstname), assign (AG.name_lastname, name.lastname)
affiliation→firstname	create(AG), assign(AG.parent affiliation iid, affiliation.iid), assign(AG.firstname, firstname.PCDATA-value)
affiliation→lastname	create(AG), assign(AG.parent affiliation iid, affiliation.iid), assign(AG.lastname, lastname.PCDATA_value)
affiliation→affiliation	create(AG), assign(AG.parent affiliation iid, affiliation.iid), assign(AG.affiliation_iid, affiliation.iid)

FIGURES 12-14

[00230] The step of loading the XML data 16 of the document 12 into the tables 20 of the relational database 14 (identified by reference number 60 in Fig. 2) will now be described in greater detail with respect to Figs. 12-13. Turning to Fig. 12, the step of loading the document 12 into the tables 20 of the relational database 14 generally involves traversing the element tree shown in Fig. 14. A simple recursive function is shown by example in Figs. 12-13 which initiates with step 246 and receives input of the type shown in the note 248 associated with step 246 in Fig. 12. This process then calls a function visit node with the arguments of (root, 0). This function essentially assists the

loader 30 in walking through the element tree. Processing then moves to step 250 where this process ends. Turning to Fig. 13, the steps performed in the `visit_node` function are shown in greater detail in which processing moves to step 252 which receives input shown in a note 254 associated with step 252 in Fig. 13. Processing then moves to decision block 256 in which it is determined whether the variable root is equal to a null value. If so, processing ends. If not, processing moves to step 258 in which a vector variable `action_vector` is set equal to a method result of the function called therein.

[00231] Processing then moves to step 260, which calls a function to execute all actions relative to the particular root and vector of the particular node 302 at issue. Processing then moves to step 262, which queries the appropriate attributes table to return a recordset containing all attributes of the particular node 302 being visited. Processing then moves to step 264, which initiates a loop for each of the attributes of the particular node and, in turn, processing moves to step 266 in which each attribute of that node is visited. Processing moves to decision block 268, which determines whether additional attributes need to be processed for this node 302. If so, processing returns to step 264. If not, processing moves to step 270.

[00232] At step 270, all of the children of the particular node 302 being visited are determined. Then processing moves to step 272, which initializes a `position` variable. Processing then moves to step 274, which initiates a loop for each child of the particular node 302 being visited. At step 276, the `position` variable initiated at step 272 is incremented. Processing then moves to step 278, which calls a function to visit each of the nodes 302 comprising children of the particular node (essentially this traverses the tree in a style shown in Fig. 14). Next, each of the links corresponding to the particular node 302 are visited in step 280. Processing then moves to decision block 282 which determines whether additional children need to be processed for this node 302. If yes, processing returns to step 274. If not processing terminates at step 284.

[00233] In general, to load an XML document 12, a created XML tree is traversed, such as that shown in Fig. 14, although this could include any tree-like structure, e.g., a parser tree in addition to the DOM model shown in Fig. 14, in depth-first and use the

pattern-mapping table 36 collected during the mapping of the DTD 18 to determine the disposition of each node 302 and its data.

[00234] All of the nodes 302 are visited to the lowest levels of the tree model 300 and all of the edges 304 are visited on the way up to the root node (number 1). For every node 302 (or link), its node 302 (or link) pattern is retrieved and the pattern mapping table 36 is queried for the corresponding actions on the tables 20 of the relational database 14. The following listing depicts pseudo-code for the loading algorithm:

```
MODULE data_loading

VARIABLES:
  PatternMappingTable plt;
  RelationalTables rt;
  XMLTree xt;

BEGIN
  VisitNode(xt.getRoot());
END

PROCEDURE visitNode
IN
  Node n;

BEGIN

  IF n is NULL
    return;
  ENDIF

GTE action from the plt based on pattern node n.
doAction(action);

FOR all the attribute a

  GET action list from the plt based on
  pattern attribute a.

  FOR EACH action
  doAction(action);
  END FOR

  FOR ALL the children d of Node n
    visitNode(d);
    visitLink(n, d);
   END FOR
END FOR

END visitNode
```

```
PROCEDURE visitLink
IN:
  Node from, to;

BEGIN

  get action list from the plt based on
  link pattern from → to.

  FOR each action
  doAction(action)
  END FOR

END visitLink

PROCEDURE doAction
IN
  Action act;

BEGIN
    IF action is create(T)
       CREATE a new tuple in that
         table T with the iid and order
    ELSE IF action is update(T.A)
       UPDATE the current tuple of the
         corresponding table T with the
         value of that attribute A.
    ELSE IF action is assign(T.A, S.B)
       UPDATE the last tuple in table T
          field A with value in the
          last tuple of table S column B.
    ELSE IF action is decompose(T.A)
       CREATE multiple tuples in the current table,
         each attribute has a single value.
    END IF

END doAction

END MODULE
```


[00235] After the pattern-mapping table 36 is generated, the pattern-mapping table 36 can be used to load the XML data 16 into the relational schema 22.

[00236] The XML document 12 shown herein relating to Example 1 and as shown as a tree structure in Fig. 14 can be used as an example of how the relational database 14 can be loaded with the data 16 including the step of traversing the node tree 300 shown in Fig. 14.

[00237] First (and with reference to Fig. 14 and the pattern-matching table 36 generated in accordance with Example 1), node 1 is encountered, which is an article type node. In response, the element pattern "article" is queried in the pattern mapping table 36. From the pattern mapping table 36, a recordset result may be returned which may include: "`create(article)`". This implicates the loading action `create` discussed earlier -- therefore, one new tuple is created in table "article" with two fields: `iid: 1`, and `order: 1`.

[00238] Next, the first child of node 1 is visited, which is of type "`title`". Again, the pattern matching table 36 is queried once again which would return the pattern "`create(title)`". A new tuple is thereby created in table "`title`", with two fields: `iid: 1`, and `order: 2`. Then, the child of this node is visited, which is node `3` of type `PCDATA`. Again the pattern mapping table 36 is queried, and, responsively, a new tuple is created in the `PCDATA` table with three fields: `iid: 1, order: 3`. Then, any value attributes of the node are visited. By again querying the attribute portion of the pattern mapping table 36, the value: "XML Relation Mapping" will be filled into the "`PCDATA`" table in the column "`value`".

[00239] Then, the link between node 3 and node 2 is visited, which is the link pattern "`title→PCDATA`". The corresponding action "`assign(title.PCDATA_value, PCDATA.value)`" would be returned. Therefore, the "`value`" field of this tuple is placed in `PCDATA`, i.e., "XML Relation Mapping" into the "`title`" table, so that the field in table "`title`" is updated with the value "XML Relation Mapping".

[00240] Then, the link between node 2 and node 1 is visited, upon which the link pattern "`article→title`" would be returned from the pattern matching table 36. The corresponding action is "`assign(article.title, title.PCDATA_value)`". Hence, the "`title`" field in the tuple in "`article`" table is updated with the value "XML Relation Mapping". Of course, the loading of the remainder of the nodes in the DOM tree 300 in Fig. 14 would follow in due course to load the entire contents of the data 16 in the document 12.

[00241] It should be noted that, because the elements of the data are the basic units in the XML document 12, the system 10 should still store the data of the corresponding elements into their tables during the loading process. However, in the case of inline attributes, if the element is in-lined into an attribute, then the table of that element is no longer used after the loading. Therefore, those unused tables could be deleted after loading the data. The following table shows the result of the data loading in the relational tables:

book

iid	Order	booktitle	GI_iid
1	33	the XML Handbook	1

article

iid	Order	title	contactauthor.idd
1	1	XML Relation Mapping	1

editor

iid	Order	name
1	32	Patti Guerrieri

contactauthors

iid	Order
1	48

G3

iid	parent_editor_idd	book.iid	monograph_idd
1	1	1	

Monograph

iid	order	title	author.iid	editor_idd
1	23	Repository Support for Metadata-based Legacy Migration	4	1

author

iid	order	id	name_firstname	name_lastname	parent_GI_idd
1	4	xz	Xin	Zhang	
2	10	gm	Gail	Mitchell	
3	16	wl	Wang-chien	Lee	
4	26	sh	Sandra	Heiler	

5	36	cg	Charles F.	Goldfarb	1*
6	42	pp	Paul	Prescod	1*

*This field is filled with the techniques of handling multi level grouping that is not addressed in this application.

G2

iid	parent_article.iid	author.idd	affiliation.idd
1	1	1	
2	1	2	
3	1	3	
4	1		1

contactauthors.authorIDS

iid	Value	Parent_contactauthors_idd
1	xz	1
2	g	1
3	ml	1

affiliation

iid	Order
1	22

G1

iid	editor.idd
1	

AG

idd	PCDATA. value	book-title	title	first-name	last-name	name. first-name
1						

AG (continued)

name. last-name	parent-related-work.idd	book .idd .idd	contact-authors .idd	mono-graph .idd	editor .idd	author .idd
	1			1		

[00242] The following summarizes the metadata tables 34 used and described in this application and discusses these tables as they relate to the invention.

Table	Description
	Storing Original DTD
DTDM-Item	Stores the Elements and Groups of the original DTD.
DTDM-Attribute	Stores the Attributes of the Elements or Groups of the original DTD.
DTDM-Nesting	Stores the Nesting relationships of the original DTD.

	Storing Converted DTD
IM-Item	Stores the Elements and Groups of a converted DTD.
IM-Attribute	Stores the Attributes of the Elements or Groups of a converted DTD.
IM-Nesting	Stores the Nesting relationship of a converted DTD.

	Storing Table Schemas
TS-JC	Stores the Join Constraint information

	Keeping Track of Mapping from Original DTD into Converted DTD
Pattern	Keeps the patterns.
Pattern-Mapping	Keeps the mapping from patterns on the table schema.

[00243] By way of further summary, the present invention contemplates an XML to a relational database mapping approach. This approach is based on storing the DTD 18 into metadata tables 34 and manipulating these tables 34 to generate metadata tables describing a relational schema, and then to generate the relational schema 22 therefrom. Several techniques have been discussed, e.g., identifying sets, inline attributes, and to identify more aspects of XML document generation, and to refine the metadata generated therein. Benefits of the present invention include:

[00244] *Integrity.* The DTD 18 is stored in metadata tables 34: This ensures an integrity constraint when modifying the DTD 18.

[00245] *Simplicity.* The automatic loading of an XML document 12. The pattern-mapping table 36 keeps track of the items and links during all of the steps of refinement of the metadata. Hence, the loading process can load the XML document 12 directly based upon the pattern-mapping table 36.

[00246] *Capture of semantics and more user-friendly query interface.* The identifying sets articulating refinement further expresses multiple-value attributes into tables, in which a user can access each value instead of being able to access the whole value. For example, by breaking the `IDREFS` type attribute into tables with an `IDREF` typed column, normal joins can be performed to determine the referenced elements. Not only does this approach have the benefits of using an element type as a table name, but also the inline attribute refinement determines the extra attributes of some element types, which were originally treated as a single element type. For example, `booktitle` becomes the attribute of `book`, instead of two tables titled `booktitle` and `book` connected by joins. Not only does this keep better semantics in the mapping, but also in the refinement of the mapping. The metadata thereby has been further improved for a better query interface (e..g, being able to query `title` instead of `title.PCDATA_value`).

[00247] *A reusable and scalable method.* The mapping approach is anticipated to be useful for the reuse of DTDs and XML documents that have many different DTDs since this mapping approach is performed by the queries on the metadata tables.

[00248] *Flexibility.* Different XML relational mapping could be represented by manipulations on the metadata tables. Hence, the inventive approach extends the automatic loading described herein to work for different kinds of mapping algorithms and even for different types of databases.

[00249] *Extensible.* The present invention has a great deal of potential than merely for mapping and loading of data, as it could also be used for optimizing query performance, extending the query capability on the XML document, reconstructing the XML data for a different DTD, and integrating with other XML data or relational data.

[00250] It has been determined by the applicants that a DTD can specify important properties of an XML document including: grouping, nesting, occurrence, element referencing, etc. In order to capture these rich relationships between elements into the relational schema of a relational database, the metadata model proposed herein consists of item, attribute, and nesting relationships. The inventive mapping approach, based on the metadata tables generated herewith, can successfully capture the relationships between

elements into constrains in a relational database. For example, the nesting properties are captured as foreign key constraints between different tables. The metadata approach also makes the automatic loading of XML documents into relational tables possible and quite easy.

[00251] While the invention has been specifically described in connection with certain specific embodiments thereof, it is to be understood that this is by way of illustration and not of limitation, and the scope of the appended claims should be construed as broadly as the prior art will permit.